Maastricht University

LAW AND TECH LAB

Legal Network Analysis

Version 0.1

Do not cite

Gustavo Arosemena

Gijs van Dijck

Roland Moerland

Maastricht, November 1, 2023

Contents

1 Network Analysis: An Introduction **5**
- 1.1 What Is Network Analysis? . . . 5
- 1.2 Origins of Network Analysis . . . 6
- 1.3 What Questions Can Be Answered with Legal Network Analysis? . . . 7
- 1.4 Key Concepts of Network Analysis . . . 8
  - 1.4.1 Nodes & Edges . . . 8
  - 1.4.2 Undirected Graphs versus Directed Graphs . . . 9
  - 1.4.3 Network Visualization . . . 9
  - 1.4.4 Bipartite Graphs . . . 9
  - 1.4.5 Graphs versus Networks . . . 10
  - 1.4.6 Directed Acyclic Graph (DAG) . . . 11
  - 1.4.7 Power Law Distribution . . . 11
  - 1.4.8 Node Degree and Node Centrality . . . 12
  - 1.4.9 Community Detection . . . 14

2 Network Properties **15**
- 2.1 Paths, Shortest Paths, and Distance between Nodes . . . 15
  - 2.1.1 When nodes are disconnected . . . 16
  - 2.1.2 Fully connected networks . . . 17
  - 2.1.3 Shortest paths . . . 17
  - 2.1.4 Random paths . . . 18
  - 2.1.5 Weighted edges . . . 19
  - 2.1.6 Network Density . . . 24
  - 2.1.7 G) Eccentricity and Network Diameter . . . 25
- 2.2 Subgraphs . . . 26
- 2.3 Flattening Bipartite Networks . . . 28
- 2.4 Network Metadata . . . 30
- 2.5 Meta-data other than weights . . . 32

3 Network Centrality **35**
- 3.1 Degree and Degree Centrality . . . 35
- 3.2 Closeness Centrality . . . 43
- 3.3 Eigenvector Centrality . . . 49
- 3.4 PageRank . . . 56
- 3.5 Betweenness Centrality . . . 61
- 3.6 Hyperlink-Induced Topic Search (HITS) . . . 70

4 Community Detection **79**
4.1 Machine Learning Methods - KMeans Clustering on Embeddings 79
4.1.1 KMeans clustering 81
4.1.2 Clustering on the adjacency matrix and the laplacian matrix 83
4.1.3 Spectral clustering 86
4.1.4 Node2Vec 92
4.2 Network Algorithms - Louvain Communities 94
4.2.1 Modularity 94
4.2.2 Louvain Communities 99
4.2.3 Centrality of communities 108

5 Data Collection and Data Preparation **113**
5.1 The Research Question Offers Guidance 113
5.2 Data Collection and Qualitative Coding 114
5.3 Data Representation 115

6 Software **119**
6.1 Specific Purpose Network Analysis Software 119
6.2 General Purpose Network Analysis Software 120
6.2.1 Gephi 120
6.2.2 Other Software Packages 123
6.3 Packages in programming software (e.g. Python, R) 123

Chapter 1

Network Analysis: An Introduction

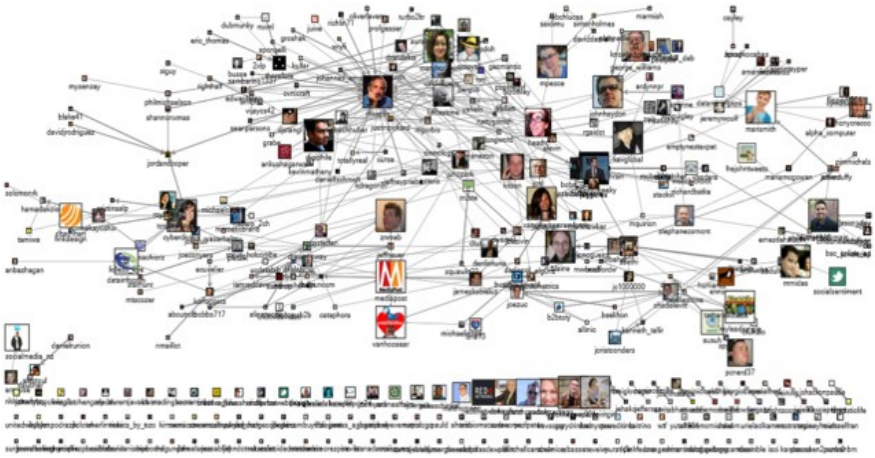
In order to run the code in this chapter, you will need to import these libraries. The *helper* library can be found in the Github repository for the book.

```
import networkx as nx
from networkx.algorithms import bipartite
import seaborn as sns
import matplotlib.pyplot as plt
from src.helper import load_graph_from_json
```

1.1 What Is Network Analysis?

Network analysis concerns the measurement and mapping of relationships between entities. Entities can consist of individuals, groups of persons, court decisions, molecules, or any other subject or object.

Applications of network analysis can be found in a variety of academic disciplines, including natural sciences, medical science, social sciences, humanities, and law. For instance, one can map communication of a group of social media users (Figure 1). By mapping who communicates with whom, clusters may emerge of groups of users who more frequently communicate with entities within their cluster than with users outside of their cluster. It can also become apparent who is more central (popular) within the various clusters or in the network as a whole. Furthermore, one may identify so-called 'brokers', individuals who connect clusters of users and consequently form the glue that holds the network together.

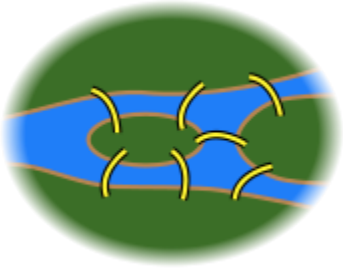


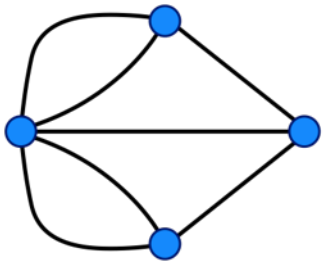
Network analysis relies on the assumption that the structural relationships between entities (e.g. persons) provide relevant information that the attributes of those entities alone cannot offer. The extent to which users are communicative, possess electronic devices, or are member of social platforms does not provide any or sufficient insight into how the users relate to one another. Network analysis reveals those relationships between entities, hence generating relevant information that the features such as communication skills or platform membership alone do not offer.

1.2 Origins of Network Analysis

Network analysis comes from graph theory, for which the foundations were laid by Leonhard Euler in 1736, who proved that the problem of 'The Seven Bridges of Königsberg' (currently Kaliningrad, Russia).

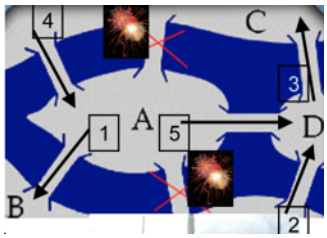
Königsberg was divided by the Pregel River, and it had two islands that were connected to each other and to the city's mainland by seven bridges. The mathematical problem that Euler solved was to design a walk through that would include the two parts of the mainland as well as the two islands where the person would cross each of the bridges exactly once.





Euler demonstrated that the 'Seven Bridges of Königsberg' problem does not have a solution, and that a walk through town while crossing every bridge exactly once, depends on the number areas (mainland + islands) and on the number of bridges. Euler proved that the desired walk can only take place if the areas are connected and include exactly zero or two areas with an odd number of bridges. Königsberg had four areas and seven bridges at the time. Three areas had three bridges and one area had five bridges. As a result, the rule is violated, as there are four areas with an odd number of bridges.

Seven bridges were bombed in World War II. Five bridges were rebuilt. Interestingly, in terms of graph theory, a Eulerian path became possible after the rebuilt (1->2->3->4->5, Figure 4). Now, two areas (B + C) have two bridges, whereas the other two areas (A + D) have three bridges (edges). This means that there are exactly two nodes with an uneven number of bridges.



1.3 What Questions Can Be Answered with Legal Network Analysis?

Network analysis focuses on relational patterns and structures that arise from interaction between the entities (called 'nodes). This approach allows analyzing a great variety of networks, ranging from online cyber communities and corporate relations networks to social movements, political affiliations, sports clubs and scholarly communities. Although there are many different categories of networks, they can more generally be grouped under the headings of technological networks (distribution, transportation, internet) information networks (citation, discourse), and social networks (friends, professional) (Newman 2018).

Network analysis can also be a relevant approach for legal scholars and criminologists. It allows legal scholars to, for instance, analyse legal citation networks. For criminologists, the network analytical approach can be relevant when conducting research on organized crime syndicates and how they are organized.

Network analysis in the legal domain is often applied to court decisions or legislation. In this regard, examples of questions are:

- What are the most important precedents? Example publication: M. Derlen and J. Lindholm (2014). Goodbye van Gend en Loos, Hello Bosman? Using Network Analysis to Measure the Importance of Individual CJEU Judgments. European Law Journal, 20(5), 667-87. https://doi.org/10.1111/eulj.12077.
- How does case importance change over time? Example publication: M. Derlen and J. Lindholm, 'Is it Good Law? Network Analysis and the CJEU's Internal Market Jurisprudence' (2017) 20 Journal of International Economic Law 257-77. https://doi.org/10.1093/jiel/jgx011
- Have certain legal topics or legal concepts gained or lost importance over time? Example publication: J. Fowler and S. Leon, 'The Authority of Supreme Court Precedent' (2008) 30 Social Networks 16-39. https://doi.org/10.1016/j.socnet.2007.05.001
- Which clusters of decisions can be distinguished? Example publication: Y. Lupu and E. Voeten (2011). Precedent in International Courts: A Network Analysis of Case Citations by the European Court of Human Rights. British Journal of Political Science, 42, 413-49. https://doi.org/10.1017/S0007123411000433

There are also studies that do not target court decisions or legislation, but are still connected to the legal field (and neighboring disciplines):

- Patent studies, for instance whether a certain inventor (e.g. Bill Gates) has invented in a wider variety of areas than another inventor (e.g. Mark Zuckerberg).
- Example publication: R. Whalen, A. Lungeanu, L. DeChurch, and N. Contractor (2020). Patent Similarity Data and Innovation Metrics, Journal of Empirical Legal Studies 17(3), 615-639. https://doi.org/10.1111/jels.12261
- How can network analysis by used to provide relevant timely and actionable intelligence for criminal networks? Example publication: P. Seidler & R. Adderley (2013). Criminal Network
- Analysis inside Law Enforcement Agencies: A Data-Mining System Approach under the National Intelligence Model. International Journal of Police Science & Management, 15(4), 323–337. https://doi.org/10.1350/ijps.2013.15.4.321.
- How do law firms partners exercise control over others in the firm? Example publication: E. Lazega, D. Krackhardt (2020). Spreading and Shifting Costs of Lateral Control Among Peers: A Structural Analysis at the Individual Level. Quality & Quantity 34, 153–175. https://doi.org/10.1023/A:1004759418226.

From these examples, it should be noted that legal network analysis can be used for empirical or computational studies of both positive law (law as it exists in codes and decisions), and of social phenomena that are relevant to law (criminal networks, corporate influence, etc.).

1.4 Key Concepts of Network Analysis

1.4.1 Nodes & Edges

Networks consist of two key concepts: nodes and edges. Nodes are the smallest unit in a network. As indicated above, nodes can consist of various types of entities: individuals, court cases, documents, words, etc. Edges are the links between two nodes. In graph representation, edges are presented

with a line (in undirected graphs) or by an arrow (in directed graphs) that points from one node to another.

Note that in the literature nodes can also be called vertices (singular: vertex) and that edges are referred to as lines, arcs (in directed networks) or edges (in undirected networks).

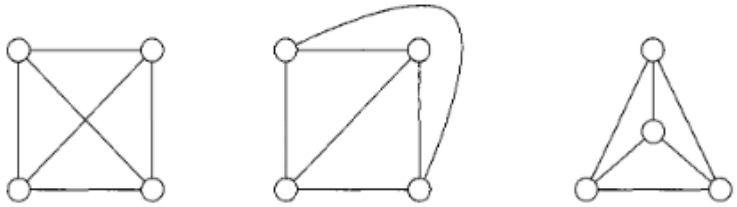
1.4.2 Undirected Graphs versus Directed Graphs

Graphs can be directed or undirected. Directed can be described as a non-reciprocal relationship (Figure 5). For instance, A sends a message to B, C admires D, or E cites F.

Undirected graphs consist of reciprocal relationships (Figure 6). A and B are friends (or: A is friends with B, B is friends with A), or A shakes hands with B. It is not common (although perhaps possible) that a friendship goes from A to B but not from B to A, making it reciprocal.

1.4.3 Network Visualization

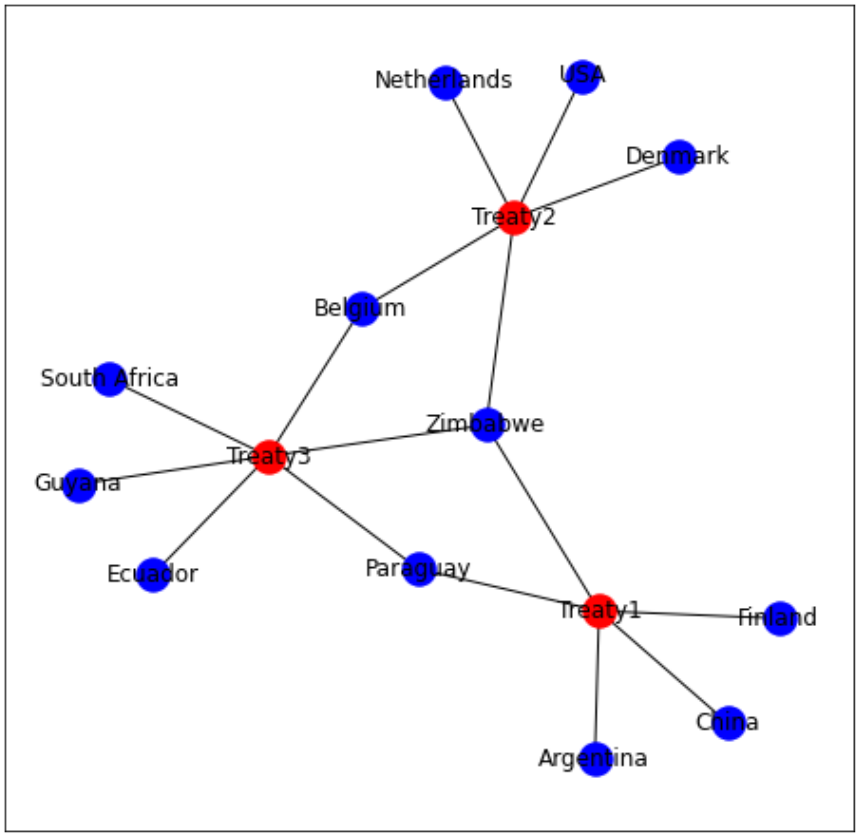
We can make observations and inferences by looking at a network. However, when we later ask ourselves questions such as "which are the most central nodes in the network?", we should not lose track of the fact that networks can be drawn in different ways without changing their properties. Some visualizations can even be misleading. The three plots below represent the exact same network (image from Wallis 2007, page 7). If one asks "which node is the most central one?" in relation to this particular network, the drawing on the left probably provides us with the best visualisation to provide the correct answer.



1.4.4 Bipartite Graphs

Networks typically deal with relations between a single class of elements. With single class of elements, we mean that the nodes are of a similar type, belong to the same class. For example, we may be interested in citations between cases, friendship between persons, and so forth. In these instances, the classes of the elements are the same: cases, persons, etc. Yet sometimes it is interesting to consider the relationship between two different types of elements. Imagine we want to research what happens when states ratify certain treaties. This must be a bipartite network because we have a relationship between two types of nodes: states and treaties. Below you can see an example of how a bipartite graph might look.

```
g_treaties = load_graph_from_json("data/g_treaties.json")
states = [x[0] for x in list(g_treaties.nodes(data="bipartite")) if x[1] == 0]
treaties = [x[0] for x in list(g_treaties.nodes(data="bipartite")) if x[1] == 1]
plt.figure(figsize=(8,8))
pos = nx.spring_layout(g_treaties, seed=123)
nx.draw_networkx_nodes(g_treaties, pos=pos, nodelist= states, node_color='blue')
nx.draw_networkx_nodes(g_treaties, pos=pos, nodelist= treaties, node_color='red')
nx.draw_networkx_edges(g_treaties, pos=pos)
nx.draw_networkx_labels(g_treaties, pos=pos);
```



1.4.5 Graphs versus Networks

Graphs and networks are common terms in the area of network analysis. Strictly speaking, the terms point to different aspects. Graphs consist of a set of nodes and a set of edges between pairs of nodes. A networks consists of a graph plus additional information regarding the nodes. These attributes can include a variety of information, such as, in case the nodes consist of documents, the year in which the document was published, the language of the documents, the creator or author, etc.

1.4.6 Directed Acyclic Graph (DAG)

Graphs that deal with legislation are often (but not necessarily) DAGs. They are directed, because the edges go from one node to another. They are acyclic, because edges commonly do not go, for instance, from A to B to C to A.

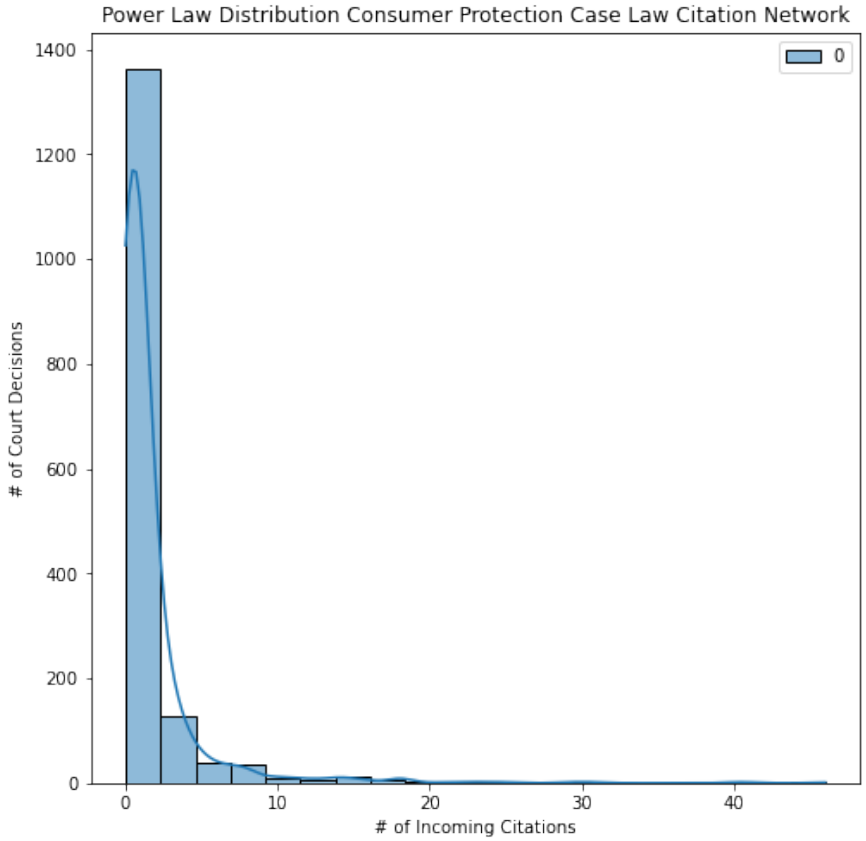
References in case law also serve as a good illustration. A citation network that consists of a number of court decisions that refer to one another is directed: the citation goes from one court decision to another. The network is also acyclic. This is because court decisions commonly only cite cases from the past, not from the future. Would a network be cyclic, it would be possible that a 2016 decision cites a court decision from the year 2020. Exceptions can, however, occur. For example, cases that are published on the same date can reference each other. In an instance of A cites B and B cites A, a loop is created, which makes the graph cyclical.

1.4.7 Power Law Distribution

Citation networks frequently have a Power Law distribution. This means that the frequency distributions are highly skewed: few nodes have a relatively high number of citations, whereas most nodes have a low number of citations. We illustrate this by means of a network of CJEU (Court of Justice of the European Union) case law, where the source nodes consist of cases that are labeled as 'consumer protection'. With source nodes, we mean the cases that were searched and for which the citations in those cases were harvested. In this network, the cases are the nodes and the references in and to the cases the edges. The network consists of 1,614 nodes (cases) and 2,662 edges (references).

Focusing on the number of incoming citations, we plot the distribution of incoming citations among the cases. We do so by means of a histogram, where the horizontal axis shows the number of incoming citations and the vertical axis the number of cases. The results reveal that a relatively small number of cases have a relatively high number of incoming citations, whereas there are a lot of cases that are hardly ever, if at all, cited. The differences are substantial: most cases are hardly cited, whereas few cases are cited over 10 times.

```
g_consprot = load_graph_from_json("data/g_consprot.json")
plt.figure(figsize=(8,8))
plt.title("Power Law Distribution Consumer Protection Case Law Citation Network")
plt.xlabel("# of Incoming Citations") #add label
plt.ylabel("# of Court Decisions") #add label
sns.histplot(dict(g_consprot.in_degree).values(), stat="count", bins=20,␣
↪kde=True);
```



One of the effects of a Power Law distribution is that it is coupled with preferential attachment. This is also called the 'rich get richer' effect: the nodes with many edges are likely to receive more edges (eg citations) in the future for the mere fact that they already had many edges. Do note that "law" in power law distributions refers to something akin to a "law of nature". It does not have anything to do with the law as in norms or rules.

1.4.8 Node Degree and Node Centrality

Network analysis is a method that can capture how central a node is in a network. This centrality can be an indicator for, for instance, the popularity or relevance of a node in a network.

Various measures exist to measure the centrality of a node. Some basic measures in legal network analysis are: - Degree. The total number of edges connected to a node. Degree Centrality normalizes (standardizes) the degree score, resulting in a score between 0 and 1 (by dividing the degree by the number of nodes minus 1 (n-1)). - Out-Degree. In directed networks, it is possible to calculate the number of edges pointing out of a node. Out-degree centrality is the normalized score of out-degree. - In-Degree. Directed networks also allow calculating the number of edges pointing to a node. In-degree centrality is the normalized score of in-degree. - Relative in-degree centrality. This metric provides more weight to more recent nodes. Older nodes are more likely to have incoming edges than more recent nodes. For instance, a court decision from 1998 is more likely to receive many

citations than one from 2021 (all other things equal), simply because it has been around longer and consequently has had more opportunities to receive citations. Relative in-degree centrality corrects for this by giving more weight to more recent nodes. - Betweenness Centrality. This score expresses the relative control that a node exerts over the flow of information within a network, i.e. number of geodesic paths that run through the node, with a geodestic path being the shortest path between two nodes. A simpler way of understanding betweenness centrality is by using the metaphor of glue: what is the glue that keeps the network together? Nodes with a high betweenness centrality score are more of a glue than nodes with a low score - without these nodes, the network is likely to fall apart or to reach another node in a 'quick' way. Another way of describing nodes with a high betweenness centrality is to call them 'brokers' that connect nodes.

We will discuss different metrics in more detail in subsequent chapters. For now, it is sufficient to have a general understanding of what the various degree and centrality scores measure and how they differ from one another.

Exercise To test whether you properly understood some of the centrality and degree measures, we turn to the so-called Krackhard Kite Graph (Figure 7). Figure 7: Krackhard Kite Graph

Answer the following questions and write down the answers for yourself. You will see the answers below.

1. Is this a directed or undirected graph?

2. Which node(s) has (have) the highest degree centrality? (incoming and outgoing links)

3. Which node(s) has (have) the highest in-degree centrality? (incoming links)

4. Which node(s) has (have) the highest betweenness centrality? (Or: What is the glue that keeps the network together / Which node fulfills the broker function the best?)

Answers

1. Is this a directed or undirected graph?

In a directed graph, you would expect the edges to have arrows, indicating from and to which node the edge goes. Our graph does not have arrows, hence we can assume it is an undirected graph.

2. Which node(s) has (have) the highest degree centrality (i.e. normalized score of incoming and outgoing edges combined)?

A count of the number of edges for each node makes it clear that Node 3 has the highest degree (degree = 6). We can calculate the normalized score by deviding the degree score by the number of nodes minus 1, which results in (6 / (9-1) =) 0.75. However, if we know which node has the highest degree score, it will also be the node with the highest degree centrality.

3. Which node(s) has (have) the highest in-degree (i.e. incoming edges)?

This is a trick question. Calculating he in-degree score is only possible in case of a directed network. Considering we have an undirected, nodes do not have an in-degree value (or an out-degree value, for that matter).

4. Which node(s) has (have) the highest betweenness centrality? (Or: What is the glue that keeps the network together / Which node fulfills the broker function the best?)

Node 7 has the shortest path to all other nodes in the network. This node therefore has the highest betweenness centrality.

1.4.9 Community Detection

Network analysis allows for the detection of communities. Some also use the terms 'cliques' or 'clusters', although these terms have slightly different meanings for some. The idea behind community detection is to group together nodes that have edges among one another. Nodes are more likely to be connected if they are members of the same community, and less likely to be connected if they are part of different communities. Nodes that belong to the same community are likely to share common attributes or functions, and they often possess different properties than the larger network (Figure 8).

Various algorithms exist to detect communities. One that is frequently used, is modularity maximization. Modularity maximazation concerns the fraction of edges within a group minus the fraction of edges that could be expected if the graph was a random graph. Modularity maximazation can be calculated, resulting in a Q score. The higher value of Q (min = -1, max = 1), the stronger the community structure in the network is. $Q = 0$ means a community structure not better than random, whereas $0.3 < Q < 0.7$, is associated with significant community structure.

Chapter 2

Network Properties

This chapter will provide more detail about the structure of networks.

In order to run the code in this chapter, you will need to import these libraries. The *helper* library can be found in the Github repository for the book.

```
import networkx as nx
import numpy as np
from networkx.algorithms import bipartite
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
from src.helper import draw_spring, load_graph_from_json
np.random.seed(123)
```

2.1 Paths, Shortest Paths, and Distance between Nodes

For any node of the network, it is possible to calculate the path it has to other nodes (if such a path exists) and its distance. A path is a series of steps getting from node A to node B. The distance is the number of steps, or the number of steps weighted by any relevant weight metric.

Some complications deserve mention here:

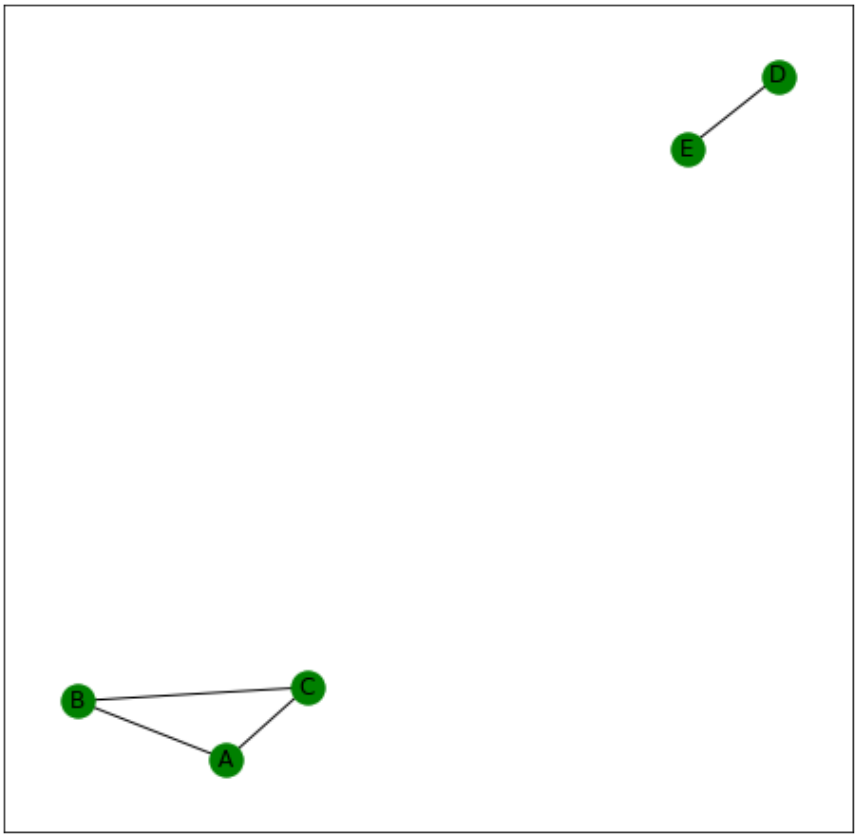
- When nodes are disconnected
- Fully connected networks
- Shortest path
- Weighted paths
- Random paths
- Eccentricity and network diameter

Let us discuss the complications in more detail.

2.1.1 When nodes are disconnected

It is not guaranteed that there will be a path between two nodes. It is possible that two sets of nodes are simply not connected. In this case there is no path between nodes A, B, or C, and nodes D or E, as we can see from this example:

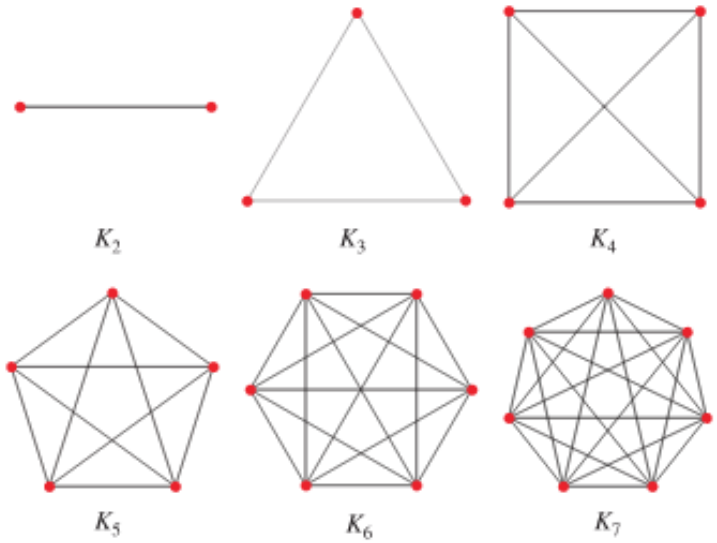
```
g_disconnected = nx.Graph()
g_disconnected.add_nodes_from(['A','B','C','D','E'])
g_disconnected.add_edges_from([('A','B'),('B','C'), ('C','A'),('D','E')])
draw_spring(g_disconnected, node_color="green")
```



When there is no path between two nodes, it makes sense to say that there is infinite distance between them. It would be getting things the wrong way round to state that there is a distance of 0. A distance of 0 would suggest that the nodes are right on top of each other, when the reality is that they are unreachable.

2.1.2 Fully connected networks

A network is fully connected if every node is connected to every other node. To see how this will look consider these graphs from Wolfram (https://mathworld.wolfram.com/CompleteGraph.html)

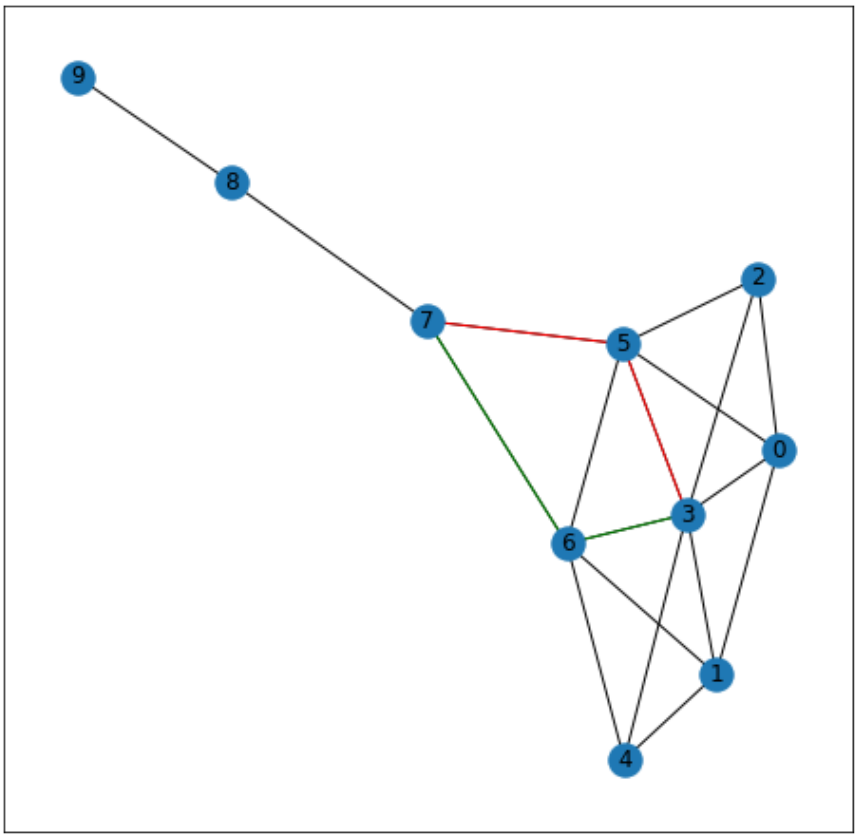


Note that for such networks it would be pointless to try to distinguish between nodes using measures like centrality or community, unless the edges have differing weights.

2.1.3 Shortest paths

The shortest path is, quite simply, the path that will reach a node in the smallest number of steps. There can be more than one "shortest path" (but all the shortest paths will have the same smallest number of steps). For example, if we look at the kite graph, there are two shortest paths from 7 to 3, one going through 6, and another going through 5. There are longer paths too, for example, 7 -> 5 -> 2 -> 3. These may not be immediately relevant, but might be interesting possible random paths, between 7 and 3.

```
g_kite = nx.krackhardt_kite_graph()
plt.figure(figsize=(8,8))
pos = nx.spring_layout(g_kite, seed=123)
nx.draw_networkx_nodes(g_kite, pos)
nx.draw_networkx_edges(g_kite, pos=pos)
nx.draw_networkx_edges(g_kite, edgelist=[(7,5),(5,3)], edge_color="red", pos=pos)
nx.draw_networkx_edges(g_kite, edgelist=[(7,6),(6,3)], edge_color="green",␣
↪pos=pos)
nx.draw_networkx_labels(g_kite, pos=pos);
```



Shortest paths are useful for many purposes. They are key ingredients in many algorithms for identifying node centrality.

While shortest paths can be easy to "see" in small graphs like this one, this will not be possible in more complex graphs. Finding the shortest path will then be a non-trivial problem. This process is automated by network analysis libraries and programs.

For the unweighted graphs that we are using, a shortest path counts discrete "steps" between nodes. This implies that all the nodes are a unit distance away. However, it is also possible to have weighted paths, as we will see in a bit.

2.1.4 Random paths

The length of the shortest path will be a definite number. However, there may be an arbitrary number of non-shortest paths wandering through the nodes. A single arbitrary path will not be of much interest (why this one?, why not another one?), but if we allow movement randomly from node to node, these random paths - random walks - can become of interest. We may be interested not only in the shortest path between A and B, but in the mean path distance between A and B, taking into account routes that go more or less directly from A to B, as well as those that make longer detours. By measuring a number of random walks for a number of nodes in the network, it

will often become clear that some nodes are more likely to be "passed through" than other nodes, which suggests these nodes are more central and perhaps therefore more relevant or interesting.

NetworkX does not have a random path function built in. However one can make one with the neighbors attribute of the Graph object. The steps are more or less like this:

1. Select a number of steps for the walk. In this case four steps. Do all the steps below until you hit four steps.
2. Select a particular node to start with, for example node 7.
3. Find all the neighbors of node 7 (in this case, 5, 6, 8).
4. Randomly choose one of these nodes to go to. Say choose node 5.
5. Update the value of your start position to the chosen node, in this case 5.
6. Record that you have made one step (3 to go).
7. If you have made less than four steps, go back to step 2. If you have made four steps, stop.

These steps can be implemented in code with some effort.

```
n = 0
start = 7
history = [start]
while n < 4:
  my_neighbors = list(g_kite.neighbors(start))
  move_to_node = np.random.choice(my_neighbors)
  history.append(move_to_node)
  start = move_to_node
  n += 1

print(history)
```

```
[7, 8, 9, 8, 7]
```

2.1.5 Weighted edges

An edge can show that there is a relationship between nodes A and B. The nature of that relationship can be many things, such as there being a train between A and B, or case A cites case B. In these instances the connection is binary: there either is a connection or there is not.

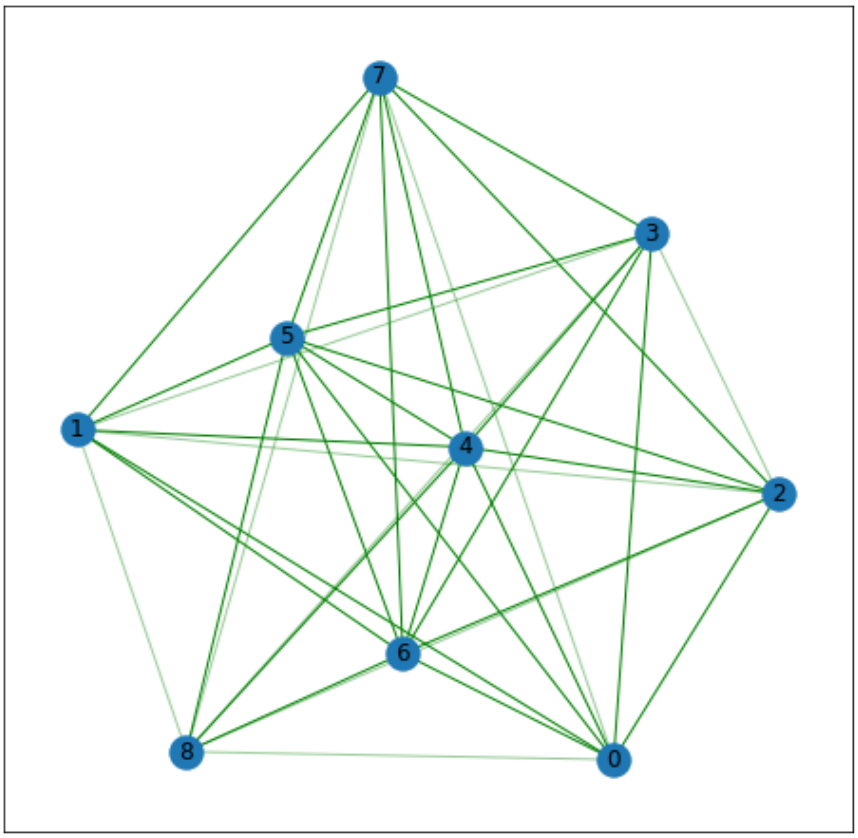
There are a range of cases where one wants to record the strength of a connection and not just its presence. For example, one might want to record, not only that there is train path from A to B, but also how long that path is in terms of kilometers. We might want to score not only that case A cites case B, but also how many times case B is cited by case A. We can add this attribute to the edge by giving weights to edges. For instance, if case A cites case B four times, the edge weight becomes four.

Weighted networks can be represented visually in an intuitive way by using different colors or line styles for their edges. Below is a network using a document similarity matrix. This matrix records how close two documents are in light of the tokens they share (Jaccard distance) and scores them with 1 if they are identical, and 0 if they are completely different. It is made of Amazon reviews of the book "The Ordinary Virtues" by Michael Ignatieff, which you can see below.

```
  Reviewer  Stars                                             Review
0      Tom      5  A really interesting perspective on the world'...
1   Merrie      5  This is a remarkable book with observations th...
2   Tatler      4  Perceptive, well-written by qualified analyst...
3   Nissim      5  The book is very important because it exposes ...
4   Regina      5                                          Very good
5      Nir      5  great book, fascinating, lucidly-presented arg...
6    David      4  This is a moving and humane reflection on mora...
7  Richard      1  The author sincerely believes that if we prole...
8    Tapas      3  It is a book about the modern globalized world...
```

In the process of comparing the distance between each document, this dataframe is turned into a matrix, and that can be used to build a network. Please see Appendix 1: "Text Similarity Networks" for more details of how this can be done. Here is one way to plot that network:

```
g_docs = load_graph_from_json("data/g_docs.json")
plt.figure(figsize=(8,8))
pos = nx.spring_layout(g_docs, seed=123)
nx.draw_networkx_nodes(g_docs, pos)
lowweight = [e for e in g_docs.edges if g_docs.edges[e]['weight'] < 0.3]
highweight = [e for e in g_docs.edges if g_docs.edges[e]['weight'] > 0.3]
nx.draw_networkx_edges(g_docs, edgelist=highweight, pos=pos, edge_color='green')
nx.draw_networkx_edges(g_docs, edgelist=lowweight, alpha=0.4, pos=pos,␣
↪edge_color='green')
nx.draw_networkx_labels(g_docs, pos=pos);
```

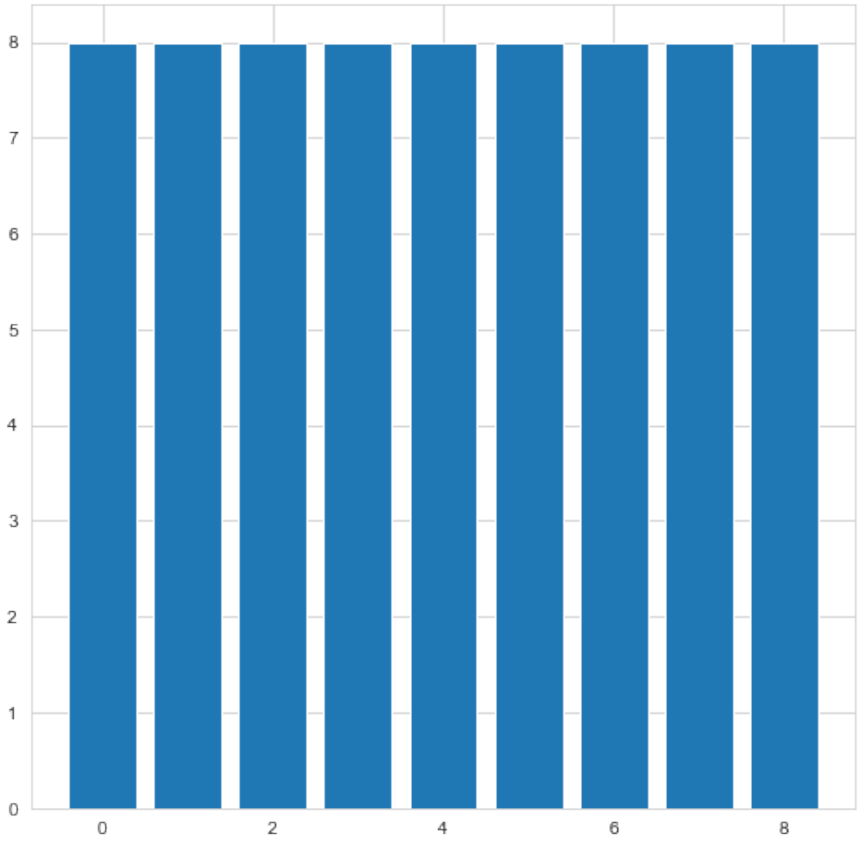


For our purposes, what is to note is that if you graph the network without consideration of weight, it gives you a fully connected network, which is a nice geometrical figure, but not very informative. Every document was compared to every other document, so every node is on step ("one degree") of separation from every other and so every node has the exact same number of neighbors.

A plot can reveal how uninformative this is:

```
plt.figure(figsize=(8,8))
plt.bar(x = list(dict(g_docs.degree).keys()), height = list(dict(g_docs.degree).
↪values()))
```

```
<BarContainer object of 9 artists>
```



It is a bit hard to see, but note that this is a fully connected or complete graph. NetworkX can reveal this to us by simply passing the graph to the function

`nx.is_connected()`

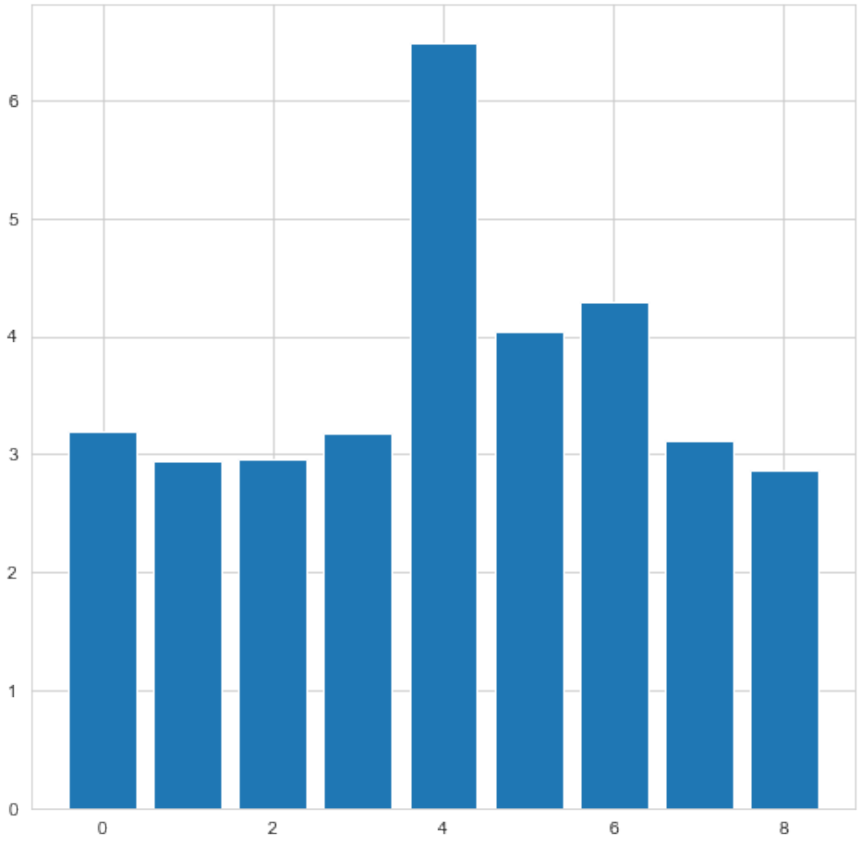
```
nx.is_connected(g_docs)
```

```
True
```

However things change if weights are added. Then one can truly get some insight into the network structure. On quick view, document 4 seems the most representative.

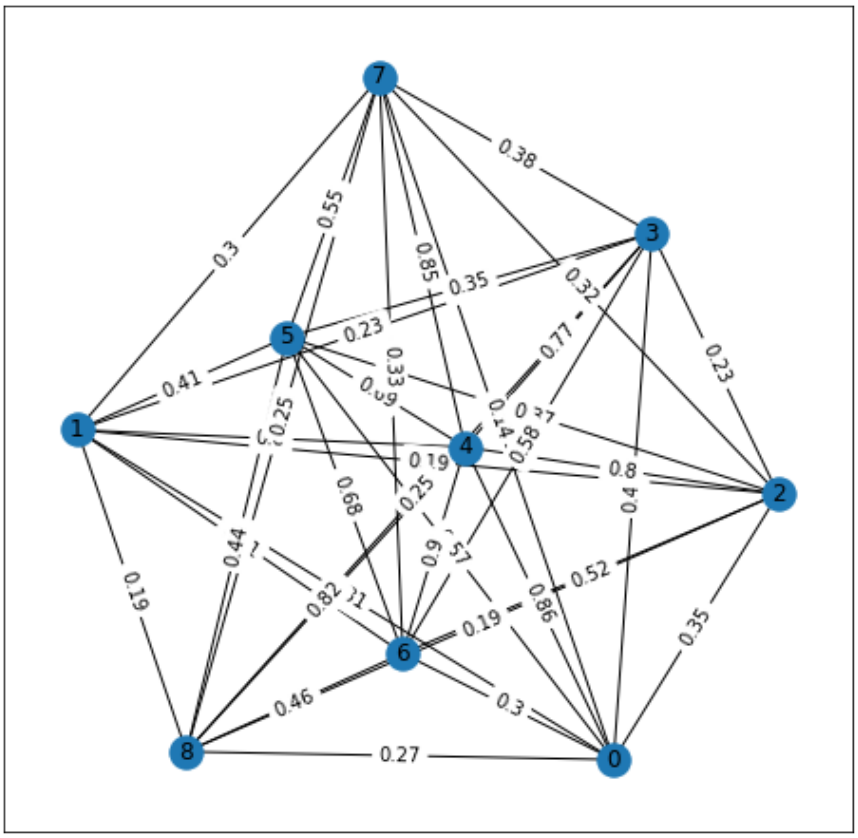
```
plt.figure(figsize=(8,8))
plt.bar(x = list(dict(g_docs.degree).keys()),
                height = list(dict(g_docs.degree(weight="weight")).values()))
```

```
<BarContainer object of 9 artists>
```



If the network is not too dense, weights can be drawn into the plot to make it more informative.

```
plt.figure(figsize=(8,8))
pos = nx.spring_layout(g_docs, seed=123)
nx.draw_networkx_nodes(g_docs, pos)
nx.draw_networkx_edges(g_docs, pos=pos)
labels = {e: np.round(g_docs.edges[e]['weight'],2) for e in g_docs.edges}
nx.draw_networkx_edge_labels(g_docs, pos, edge_labels=labels)
nx.draw_networkx_labels(g_docs, pos=pos);
```



In the past we stated that assessing centrality or community of fully connected networks can be pointless, for all the nodes are then by definition connected to all the other nodes, and so they will get identical scores. However in this case it is not pointless to assess the centrality or community of its nodes, because the edges have different weights. This is illustrated in the section on closeness centrality below.

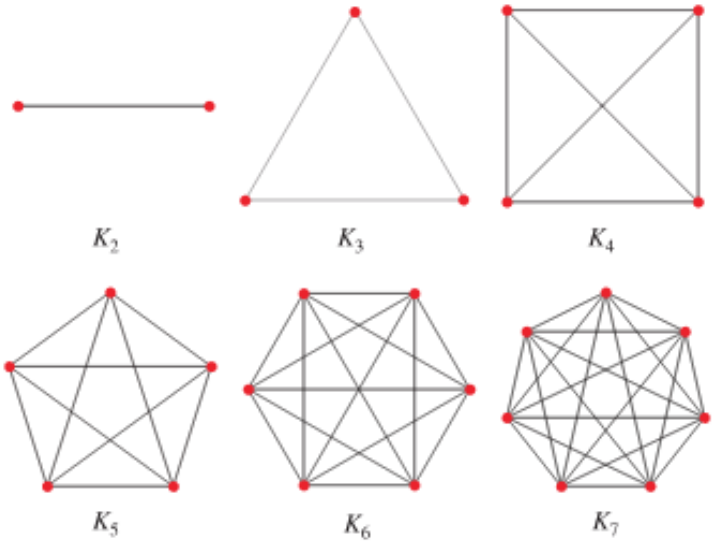
Calculating shortest paths is different for weighted edges than for unweighted edges. In unweighted graphs it would be a matter of counting the steps on different paths and finding the one with the lowest number of steps, whereas in weighted graphs the procedure is to count the steps and multiply every step by its weight.

Also note that the weight can represent closeness or distance. If it represents distance, then one must simply multiply by it. If it represents closeness, we need to convert it into distance by using its reciprocal.

$$distance = \frac{1}{closeness}$$

2.1.6 Network Density

Lets remember the fully connected networks discussed earlier



Since every node here will be connected to every other node (there are no strangers, everyone is a neighbor of everyone else, everyone has maximum popularity) it can be said that these networks are maximally dense.

A maximally dense network will have $\frac{n(n-1)}{2}$ edges.

So a network with 3 nodes can have a maximum of 3 edges, an network with 4 nodes 6 edges, a network with 5 nodes 10 edges, and so forth.

The network density will be the ratio of the number of edges actually present m to the hypothetical maximum, in other words:

$$\frac{m}{n(n-1)/2}$$

2.1.7 G) Eccentricity and Network Diameter

Next we need to consider eccentricity. Eccentricity records the longest shortest path between every node. In the kite graph above, we can see that the eccentricity of node 9 is 4, as the maximum shortest path that exist between that node and some other node is four steps. Node 7, by contrast, has a maximum eccentricity of 2, as the longest shortest path that exist between it any other node is just 2.

Just as we can be interested in what is the center of a network, we can be interested in how large the network is. However, one cannot just "eye" a network graph to get a sense of its dimensions, because a graph can be plotted in many different ways and still be the same network.

The diameter of a network is very simple to calculate. It is just the maximum eccentricity value. A network is as wide as the longest shortest path that it includes. For the kite network, no node is further away than four steps from any other (that is the longest shortest path), and thus that is its diameter.

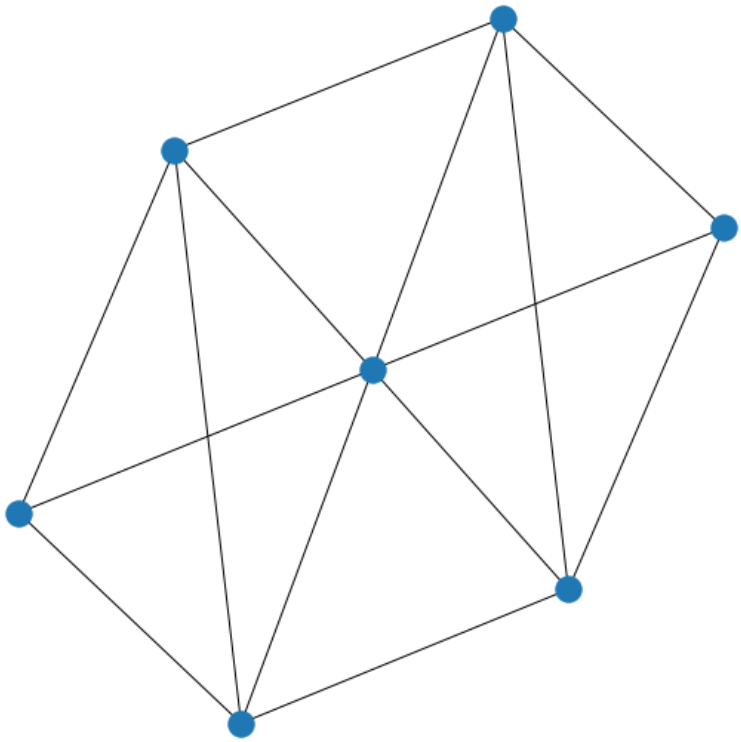
```
nx.diameter(g_kite)
```

```
4
```

2.2 Subgraphs

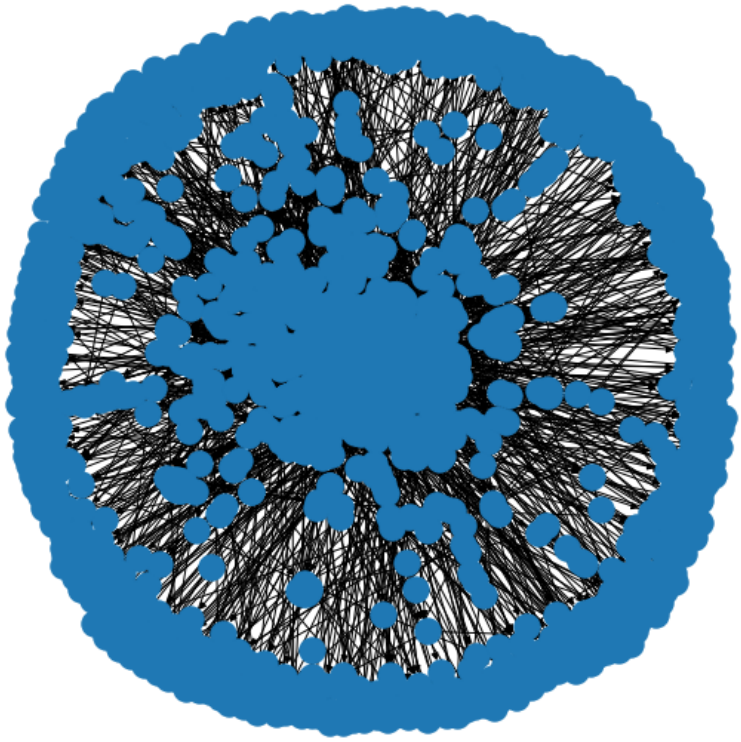
A graph can be broken down into subgraphs, which have the same (trivially) or less nodes than the original graph. This can be done by specifying the nodes you want to select. Here, for example, we take the "tail" out of the kite, by not including nodes 7, 8, and 9 in the subgraph.

```
g_subkite = g_kite.subgraph([0,1,2,3,4,5,6])
plt.figure(figsize=(8,8))
nx.draw(g_subkite)
```



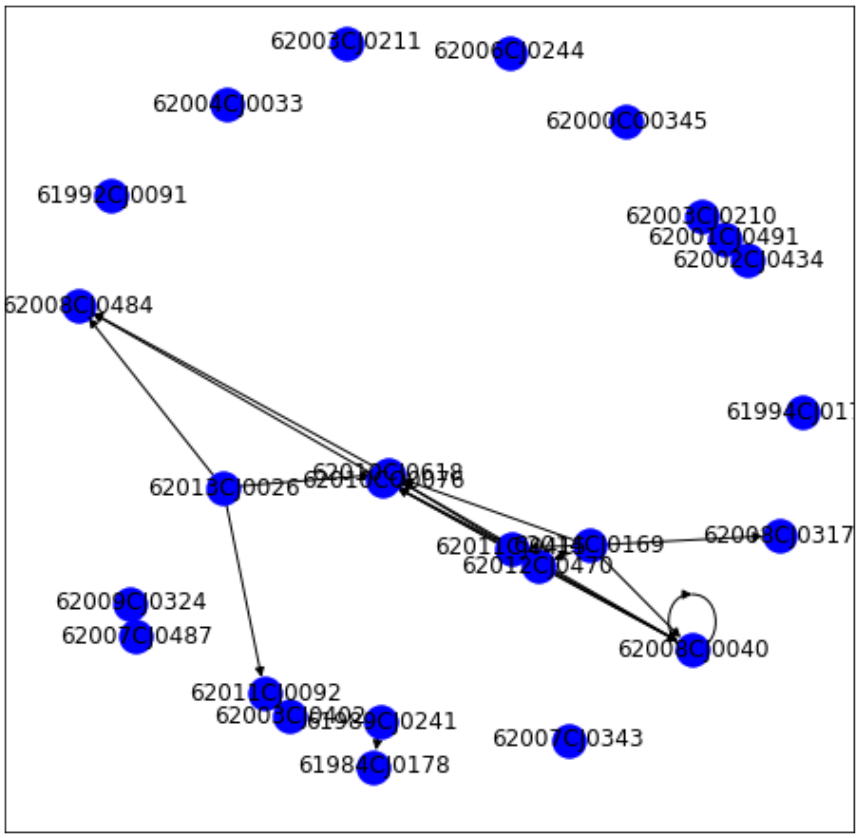
The kitegraph is a rather small graph and intuitive to interpret when visualized. The CJEU case law network is a larger one that is uninterpretable when all nodes are visualized.

```
g_consprot = load_graph_from_json("data/g_consprot.json")
plt.figure(figsize=(8,8))
nx.draw(g_consprot)
```



One (!) way of making the network more comprehensible, is to create a subnetwork. Here, we create and plot a subnetwork of cases that have a degree score of at least 25, meaning that they have 25 or more citations (counting both incoming and outcoming citations). The network becomes much smaller now.

```
node_degree_dict=nx.degree(g_consprot)
g_consprot2=nx.subgraph(g_consprot,[x for x in g_consprot.nodes() if␣
 ↪node_degree_dict[x]>=25])
# note: this code has already been run in the prerequsites section, but we are␣
 ↪running it again here to exemplify the process of
# making a subgraph
draw_spring(g_consprot2)
```



Some nodes are disconnected from the other nodes. How is that possible considering we selected the cases with a degree score of 25 or more? Note that we did not filter (mask) the network where we instructed the machine to only display the nodes in the original graph with a degree of 25 or higher. Instead, we created a subgraph from which we removed the nodes with a lower degree. When the nodes are removed, the edges to the nodes are also removed, as a result of which the degree values of the nodes in the subnetwork also decreases (because they are not or less frequently cited by the remaining nodes) to the extent that nodes can have a degree of 0.

2.3 Flattening Bipartite Networks

A bipartite network has two types of nodes. One such example already introduced is states (node type 1) and treaties that they have ratified (node type 2).

Because of its bipartite form, such network cannot be easily analyzed using conventional methods such as centrality, edges to treaties and to countries are treated the same. One solution is to project the network into a single level, that is, to flatten it.

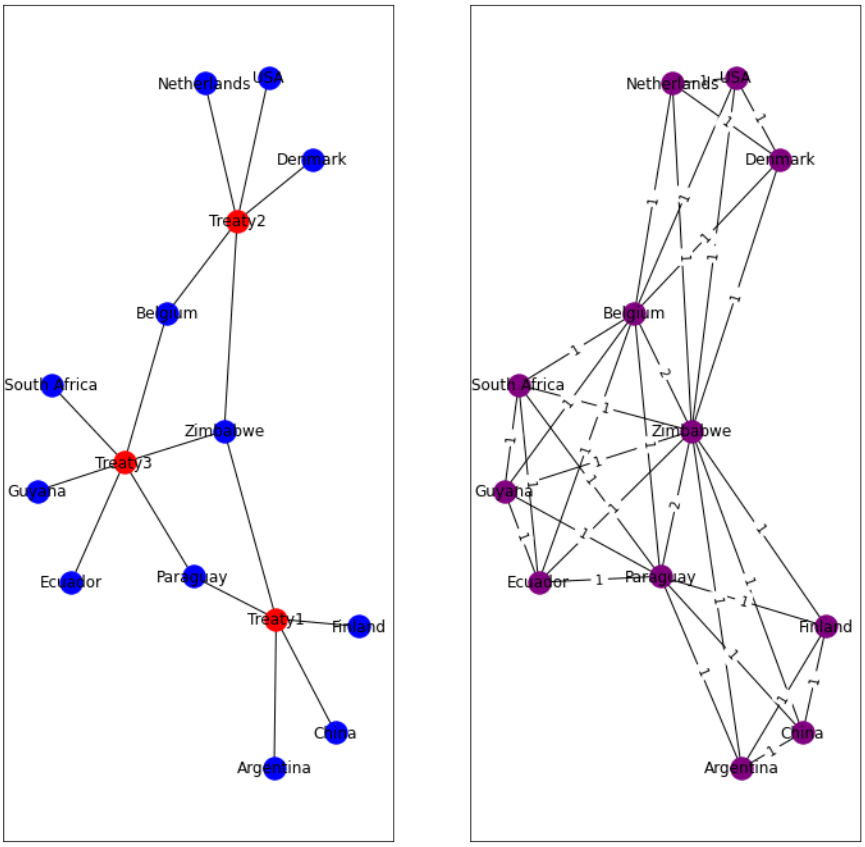
Flattening involves removing the upper level nodes (treaties). Instead one creates connections between the lower level nodes (countries), weighted by how many upper level nodes they shared before.

```
g_treaties = load_graph_from_json("data/g_treaties.json")
states = [x[0] for x in list(g_treaties.nodes(data="bipartite")) if x[1] == 0]
treaties = [x[0] for x in list(g_treaties.nodes(data="bipartite")) if x[1] == 1]
g_treaties_flat = nx.bipartite.weighted_projected_graph(g_treaties, nodes=states)
```

```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12,12))

pos = nx.spring_layout(g_treaties, seed=123)
nx.draw_networkx_nodes(g_treaties, pos=pos, nodelist= states, node_color='blue',␣
↪ax=ax[0])
nx.draw_networkx_nodes(g_treaties, pos=pos, nodelist= treaties,␣
↪node_color='red', ax=ax[0])
nx.draw_networkx_edges(g_treaties, pos=pos, ax=ax[0])
nx.draw_networkx_labels(g_treaties, pos=pos, ax=ax[0]);

edgelabels = {(e,w):list(g_treaties_flat.get_edge_data(e,w).values())[0] for e,w␣
↪in g_treaties_flat.edges}
pos = nx.spring_layout(g_treaties, seed =123)
nx.draw_networkx_nodes(g_treaties_flat, pos=pos, node_color="purple", ax=ax[1])
nx.draw_networkx_edges(g_treaties_flat, pos=pos, ax=ax[1])
nx.draw_networkx_labels(g_treaties_flat, pos=pos, ax=ax[1]);
nx.draw_networkx_edge_labels(g_treaties_flat, pos, edge_labels=edgelabels,␣
↪ax=ax[1]);
```



Unlike the previous networks that we used, which were comprehensible, this network is not even that large yet it is difficult to interpret with the naked eye. For instance, it is difficult to determine whether this network is fully connected. A computation shows this is the case.

```
nx.is_connected(g_treaties_flat)
```

```
True
```

2.4 Network Metadata

Networks can store more than just connections between nodes and edges. The nodes and the edges can have attributes. We have already encountered one of such attributes, which is the "weight" attribute on an edge. Accessing such information we can query the network and visualize subnetworks that meet specific criteria.

Lets first query edge weights using the document network that already has weights.

```
list(g_docs.edges(data="weight"))
```

```
[(0, 1, 0.3057850860342909),
 (0, 2, 0.3515643806261561),
 (0, 3, 0.4011441744419861),
 (0, 4, 0.8588650335933432),
 (0, 5, 0.5663867925847853),
 (0, 6, 0.3036944274298017),
 (0, 7, 0.1427956434168483),
 (0, 8, 0.2663586423650602),
 (1, 2, 0.1887291756697731),
 (1, 3, 0.2301873550390033),
 (1, 4, 0.800124882934437),
 (1, 5, 0.4122354734424759),
 (1, 6, 0.5097321851655097),
 (1, 7, 0.3027866619120244),
 (1, 8, 0.1868838597034725),
 (2, 3, 0.2280158058874547),
 (2, 4, 0.7995610591992457),
 (2, 5, 0.366590703499365),
 (2, 6, 0.5195230803085847),
 (2, 7, 0.317271203011655),
 (2, 8, 0.1896235506799158),
 (3, 4, 0.7678964587257362),
 (3, 5, 0.3480240462634174),
 (3, 6, 0.5762447937744228),
 (3, 7, 0.3808681395940203),
 (3, 8, 0.2481169176073587),
 (4, 5, 0.6878902741436519),
 (4, 6, 0.9008941138337112),
 (4, 7, 0.8540864919561113),
 (4, 8, 0.8214267919513534),
 (5, 6, 0.6824646015295808),
 (5, 7, 0.5453013285723745),
 (5, 8, 0.4356888455303136),
 (6, 7, 0.3289244386028924),
 (6, 8, 0.4649224309916062),
 (7, 8, 0.2451832039590595)]
```

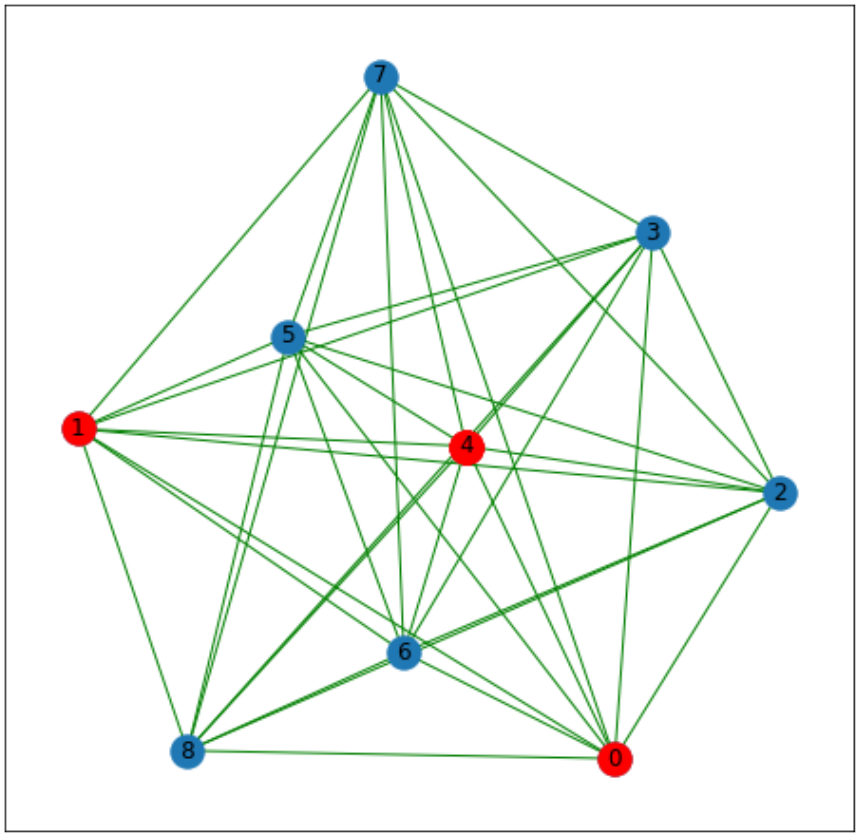
We can then filter out the edges that satisfy a certain edge condition.

For example here we will isolate the nodes that have an edge with a weight of more than 0.8 and see them as a subgraph or draw them as part of a graph with a different color.

```
mynodelist = [node for node, edge, weight in g_docs.edges(data="weight") if␣
 ↪weight > 0.8]
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(8,8))

pos = nx.spring_layout(g_docs, seed=123)
nx.draw_networkx_nodes(g_docs, pos)
```

```
nx.draw_networkx_nodes(g_docs, nodelist=mynodelist, pos=pos, node_color="red")
nx.draw_networkx_edges(g_docs, pos=pos, edge_color='green')
nx.draw_networkx_labels(g_docs, pos=pos);
```



2.5 Meta-data other than weights

In fact NetworkX can include arbitrary metadata, not just edge weights. The best way to encode such information is directly in the file format that generates the network. For this the most ideal approach might be JSON, as this puts very little limits on what you can add as additional information and be edited manually or through code. In this example we can see that some nodes in the legal history network have a "type" designator in addition to the normal node id and edge information. For more detail on JSON see appendix 1.

```
g_lh = load_graph_from_json("data/g_lh.json")
```

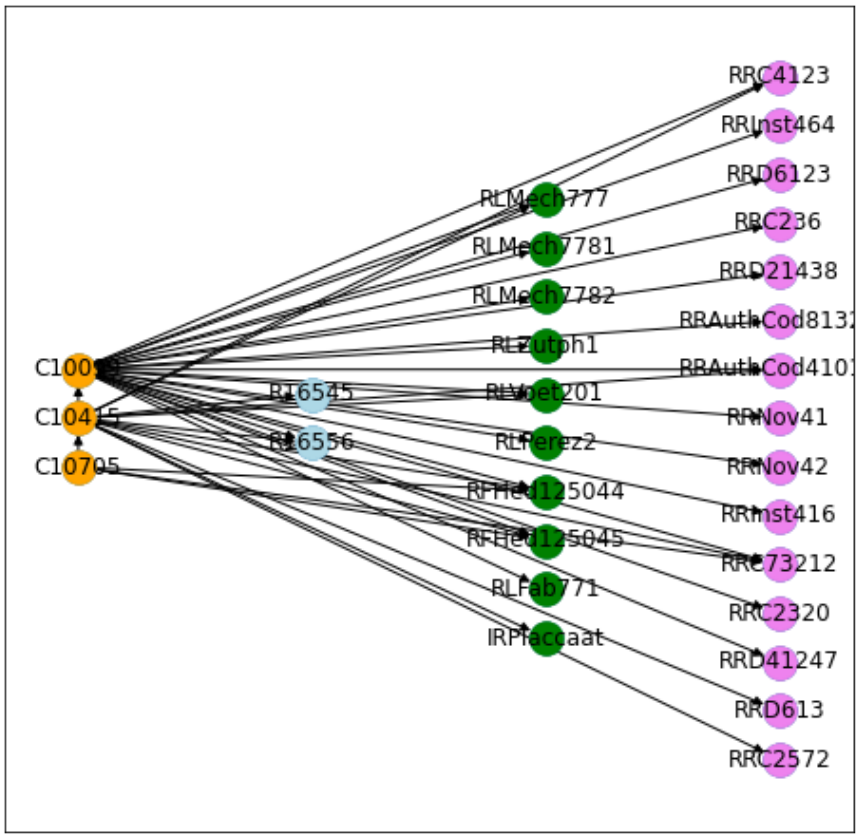
```
resdict = dict(g_lh.nodes(data="type"))
pd.DataFrame({'nodes':resdict.keys(), "values":resdict.values()})
```

	nodes	values
0	R16545	Case law
1	R16556	Case law
2	C10099	Basic source
3	C10415	Basic source
4	C10705	Basic source
5	RRC4123	Roman law
6	RLMech777	Customary law
7	RLMech7781	Customary law
8	RLMech7782	Customary law
9	RLZutph1	Customary law
10	RLVoet201	Customary law
11	RRInst464	Roman law
12	RRD6123	Roman law
13	RRC236	Roman law
14	RRD21438	Roman law
15	RRAuthCod81324	Roman law
16	RRAuthCod41014	Roman law
17	RRNov41	Roman law
18	RRNov42	Roman law
19	RLPerez2	Customary law
20	RRInst416	Roman law
21	RRC73212	Roman law
22	RFHed125044	Customary law
23	RFHed125045	Customary law
24	RRC2320	Roman law
25	RRD41247	Roman law
26	RLFab771	Customary law
27	RRD613	Roman law
28	RRC2572	Roman law
29	IRPlaccaat	Customary law

```
basic_source = [x for x, y in dict(g_lh.nodes(data="type")).items() if y ==␣
↪"Basic source"]
case_law = [x for x, y in dict(g_lh.nodes(data="type")).items() if y == "Case␣
↪law"]
roman_law = [x for x, y in dict(g_lh.nodes(data="type")).items() if y == "Roman␣
↪law"]
customary_law = [x for x, y in dict(g_lh.nodes(data="type")).items() if y ==␣
↪"Customary law"]
```

```
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(8,8))
pos = nx.multipartite_layout(g_lh, subset_key="type")
nx.draw_networkx_nodes(g_lh, pos)
nx.draw_networkx_nodes(g_lh, nodelist= basic_source, pos=pos,␣
↪node_color="orange")
```

```
nx.draw_networkx_nodes(g_lh, nodelist= roman_law, pos=pos, node_color="violet")
nx.draw_networkx_nodes(g_lh, nodelist= customary_law, pos=pos,␣
 ↪node_color="green")
nx.draw_networkx_nodes(g_lh, nodelist= case_law, pos=pos, node_color="lightblue")
nx.draw_networkx_edges(g_lh, pos=pos)
nx.draw_networkx_labels(g_lh, pos=pos);
```



Chapter 3

Network Centrality

As noted in Chapter 1, there are a variety of network centrality measures, and they capture different information. Which centrality measure to use, depends on the question one is interested in. The presentation that follows will try to give you insight into some considerations that count when choosing which measure to focus on. This will involve comparing the results of different measures on simple networks. Please note that it is not possible to compare the results of different centrality measures unless they are "normalized" first. For more on normalization please see Appendix 3.

In Chapter 1, we introduced a number of centrality measures. In this chapter, we more thoroughly discuss these measures. We illustrate by means of examples. For those who are interested, we provide for basic mathematical explanations for the centrality measures. It is also possible to skip the math and to focus on the conceptual explanations provided below.

In order to run the code in this chapter, you will need to import these libraries. The *helper* library can be found in the Github repository for the book.

```
import networkx as nx
import numpy as np
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
from networkx.drawing.nx_pydot import graphviz_layout
from src.helper import draw_spring, load_graph_from_json, score_normalize
np.random.seed(123)
```

3.1 Degree and Degree Centrality

One might be interested in finding nodes, for instance court decisions, that are connected to a high number of other decisions, for instance by means of a reference (citation). The number of references in or to a certain decision may be an indication of the precedent value of that particular decision: nodes that are connected to a lot of other nodes might be considered better precedents than nodes that have less connections. **Degree** concerns assessing centrality by counting how many neighbors a node has. It is possible to calculate Degree for every node and to subsequently rank the results

from the one with most neighbors to the one with the least. That is to say, to rank the nodes by looking at their degree: the number of nodes that are directly linked to it.

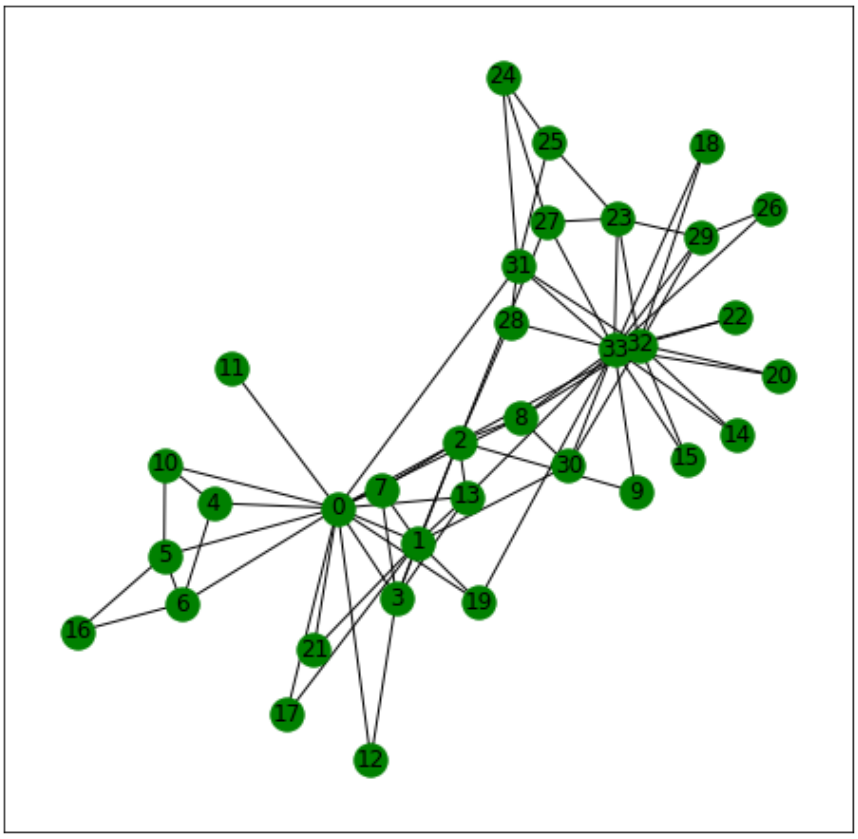
Degree provides for an absolute value. For instance, a node can have a degree score of 1, 5, 673, or any other value for that matter. However, an absolute value might not always provide relevant information. For instance, a degree score of 5 is high if the network consists of 6 nodes, yet it is likely (but not necessarily) considered to be low if the network consists of 2,000,000 nodes. Having a score between 0 and 1, regardless of the degree value and the number of nodes in the network, would make the results in the example (degree=5, nodes=6 versus degree=5, nodes=2,000,000) comparable. This is what **degree centrality** scores reflect.

Degree centrality is the simplest of all the centrality scores, and it considers only the direct number of connections you have. It does not consider things like indirect connections, or the relative importance of those who connect to you. So for those we will have to look at other measures.

To obtain the Degree score, all one needs to do is count. It might be feasible to do this by hand in relatively small networks, yet it becomes unmanageable in larger networks. Software can then do the counting for us. Degree centrality normalizes this count dividing it by the total number of nodes (with some details discussed later)

For an easy illustration of how the calculation of degree and centrality measures looks in practice, we turn to an example of a 1970s karate club. This network arises from a sociological study of a karate club that would break up, with some students following the instructor and others the club president. What we are going to do here is to guess, using degree centrality, who might these two high profile club members be based on the number of friendships each karate club member has. For this, we can calculate the Degree scores for each individual node and subsequently rank the scores in a descending order (highest scores first, followed by the lower scores). We suspect that the individuals with the highest Degree scores might be the instructor and the president.

```
g_karate = nx.karate_club_graph()
draw_spring(g_karate, node_color="green")
```




```
degree = nx.degree(g_karate)
nodeid = list(dict(degree).keys())
degscore = list(dict(degree).values())
centrality = nx.degree_centrality(g_karate)
centscore = list(dict(centrality).values())
resdf = pd.DataFrame({"node_id":nodeid, "degree":degscore, "deg_centrality":
 ↪centscore})
resdf.set_index("node_id", inplace=True)
resdf.sort_values('degree', ascending=False)
```

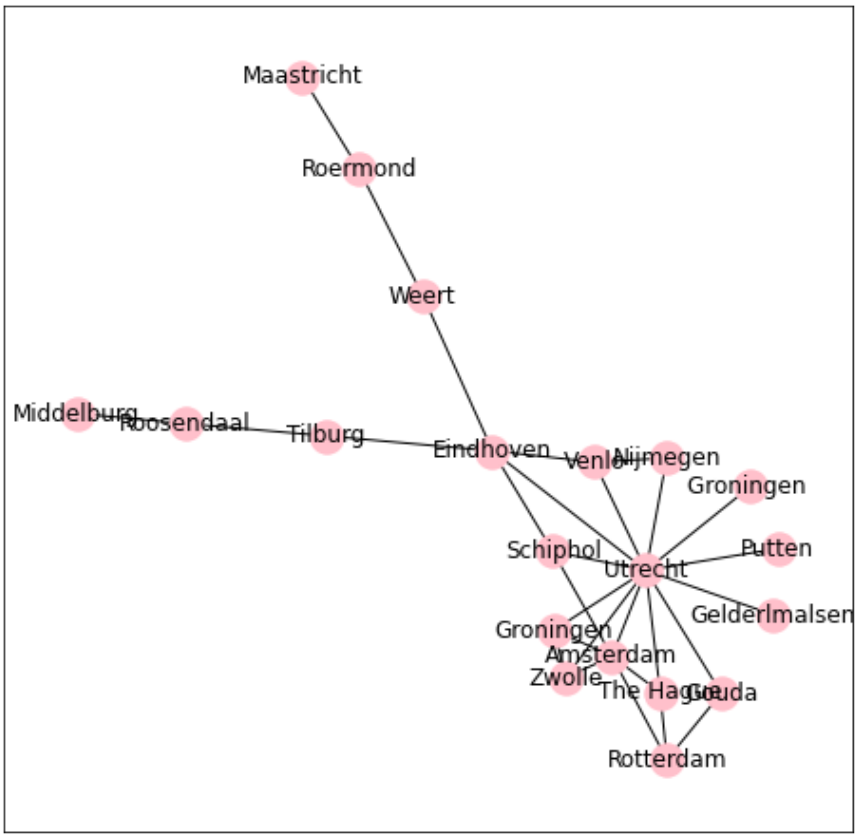
node_id	degree	deg_centrality
33	17	0.515152
0	16	0.484848
32	12	0.363636
2	10	0.303030
1	9	0.272727
3	6	0.181818
31	6	0.181818
23	5	0.151515

```
8               5       0.151515
13              5       0.151515
27              4       0.121212
29              4       0.121212
30              4       0.121212
7               4       0.121212
6               4       0.121212
5               4       0.121212
10              3       0.090909
28              3       0.090909
4               3       0.090909
19              3       0.090909
25              3       0.090909
24              3       0.090909
26              2       0.060606
18              2       0.060606
22              2       0.060606
21              2       0.060606
20              2       0.060606
16              2       0.060606
15              2       0.060606
14              2       0.060606
12              2       0.060606
9               2       0.060606
17              2       0.060606
11              1       0.030303
```

If the assumption is that club president and instructor are the individuals with the most friendships, we can look for the two nodes with the highest degree value. In the example, these are nodes 33 and 0.

We can do a similar exercise for a network of (some) train stations in the Netherlands.

```
g_trains = load_graph_from_json("data/g_trains.json")
draw_spring(g_trains, node_color="pink")
```



We can calculate the degree for each station. There are 20 train stations in this dataset. When we ask NetworkX to calculate the results, we can see that Utrecht has the highest degree (degree=12).

```
degree = nx.degree(g_trains)
nodeid = list(dict(degree).keys())
degscore = list(dict(degree).values())
centrality = nx.degree_centrality(g_trains)
centscore = list(dict(centrality).values())
resdf = pd.DataFrame({"node_id":nodeid, "degree":degscore, "deg_centrality":
 ↪centscore})
resdf.set_index("node_id", inplace=True)
resdf.sort_values('degree', ascending=False)
```

node_id	degree	deg_centrality
Utrecht	12	0.632
Amsterdam	6	0.316
Eindhoven	5	0.263
Groningen	3	0.158
Zwolle	3	0.158

The Hague	3	0.158
Rotterdam	3	0.158
Schiphol	3	0.158
Venlo	3	0.158
Nijmegen	2	0.105
Roermond	2	0.105
Weert	2	0.105
Gouda	2	0.105
Tilburg	2	0.105
Roosendaal	2	0.105
Maastricht	1	0.053
Putten	1	0.053
Gelderlmalsen	1	0.053
Middelburg	1	0.053
Groningen	1	0.053

A) Undirected degree centrality The idea of degree centrality is wholly presented above. However lets introduce some notation, and some details.

One detail is that in its basic undirected form, degree centrality is the number of edges linking to a node divided by the number of nodes minus 1 (the -1 removes consideration of the node whose centrality is analyzed. The operation has very little practical difference, and some practical implementations will not bother to subtract 1).

In terms of notation, the degree centrality of a node can be represented by the following formula:

$$\frac{degree(n)}{|N|-1}$$

Here the large N refers to the set of all the nodes in the network. The small n refers to any particular node, whereas *degree* is a function that takes a node n, and returns how many edges link to that node.

A more precise way to denote Degree Centrality would be:

$$\frac{degree(n)}{|N|-1} \mid n \in N$$

This denotation explicitly tells us that n is a node of N (the $\in$ symbol states that n is an element of the set N). The vertical bars in $|N|$ refer to the cardinality of the nodes in the network. In set theory, cardinality is simply the number of items a set has. So the cardinality of the set of colors blue, green, yellow is 3. The cardinality of the set of days in the week is 7, etc. Mathematical notation, however, is contextual. The vertical bars around N are sometimes used to mean something else (such as the vector norm), so one must interpret them in the context of what is being discussed.

Now we can put this in practice and do the formula by hand. For example looking back at the trains network, we can see that Maastricht has a degree of 1: there is only one station that links to Maastricht, in this case Roermond. So the result is simply:

$$\frac{degree(Maastricht)}{|N|-1}$$

or

$$\frac{1}{20-1} = 0.052$$

What is the degree of Eindhoven? It is connected to Tilburg, Venlo, Utrecht, Schipol, and Weert, so it has a degree of 5. This means that its centrality score should be:

$$\frac{degree(Eindhoven)}{|N|-1}$$

or

$$\frac{5}{20-1} = 0.26$$

You can check above, and see that this is the same result that nx gives you.

B) Directed (in-degree and out-degree) centrality Degree centrality considers the number of neighbors a node has, in relation to the total number of nodes. In directed networks, one can differentiate between in-degree (the number of nodes that point to a given node) and out-degree (the number of nodes that a given node points out to). In our case law network, we might expect that the number of incoming reference to a case is a good indicator or precedent value. Or we might be looking for cases that give an overview of existing case law, in which case we might want to look at the out-degree scores of the nodes, where higher scores indicate more references to other cases.

It can be useful to have a list of cases ordered by their "importance" (degree or centrality) in terms of incoming or outgoing edges. We illustrate with the case law example, where we can find the in-degree scores for the individual nodes. We can calculate the degree, in-degree, or out-degree centrality of all nodes in our case law network (ranked or unranked), but with a network of 1,614 the list can be rather long.

Lets first print out the results sorted by in degree.

```
indegree = g_consprot.in_degree(g_consprot)
outdegree = g_consprot.out_degree(g_consprot)
nodeid = list(dict(indegree).keys())
indegscore = list(dict(indegree).values())
outdegscore = list(dict(outdegree).values())

incentrality = nx.in_degree_centrality(g_consprot)
incentscore = list(dict(incentrality).values())
outcentrality = nx.out_degree_centrality(g_consprot)
outcentscore = list(dict(outcentrality).values())

resdf = pd.DataFrame({"node_id":nodeid,
                      "indegree":indegscore,
```

```
                    "outdegree": outdegscore,
                    "indeg_centrality":incentscore,
                    "outdeg_centrality":outcentscore})
resdf.set_index("node_id", inplace=True)
resdf.sort_values('indegree', ascending=False)
```

node_id	indegree	outdegree	indeg_centrality	outdeg_centrality
61992CJ0091	46	7	0.029	4.340e-03
62003CJ0210	41	31	0.025	1.922e-02
62001CJ0491	40	40	0.025	2.480e-02
61994CJ0178	31	5	0.019	3.100e-03
62008CJ0040	30	16	0.019	9.919e-03
...	...	...	...	...
62011CJ0332	0	1	0.000	6.200e-04
62011CJ0419	0	1	0.000	6.200e-04
62011CJ0215	0	1	0.000	6.200e-04
62011CJ0133	0	2	0.000	1.240e-03
62004CJ0420	0	1	0.000	6.200e-04

[1614 rows x 4 columns]

Now lets do the same for out degree.

```
resdf.sort_values('outdegree', ascending=False)
```

node_id	indegree	outdegree	indeg_centrality	outdeg_centrality
62001CJ0491	40	40	2.480e-02	0.025
62003CJ0210	41	31	2.542e-02	0.019
62006CJ0244	9	27	5.580e-03	0.017
62012CJ0470	15	24	9.299e-03	0.015
62014CJ0377	2	22	1.240e-03	0.014
...	...	...	...	...
62009CJ0185	1	0	6.200e-04	0.000
61996CJ0180	2	0	1.240e-03	0.000
62005CJ0152	4	0	2.480e-03	0.000
62004CJ0375	1	0	6.200e-04	0.000
61974CJ0008	10	0	6.200e-03	0.000

[1614 rows x 4 columns]

We can once more check manually what is going on. For this, we take the degree, in-degree, or out-degree, and divide it by the number of nodes in the network. For example:

$$\frac{indegree(62003CJ0210)}{|N|-1}$$

or

$$\frac{41}{1614-1} = 0.025$$

Below, we show the ten nodes with the highest in-degree centrality. We rank the nodes in descending order (node with the highest in-degree on top, followed by the node with the second-highest in-degree value, etc.). Note that the ranking is the same as above, where the in-degree scores were calculated. This can be explained by the fact that all in-degree scores are divided by the same value in the demoninator.

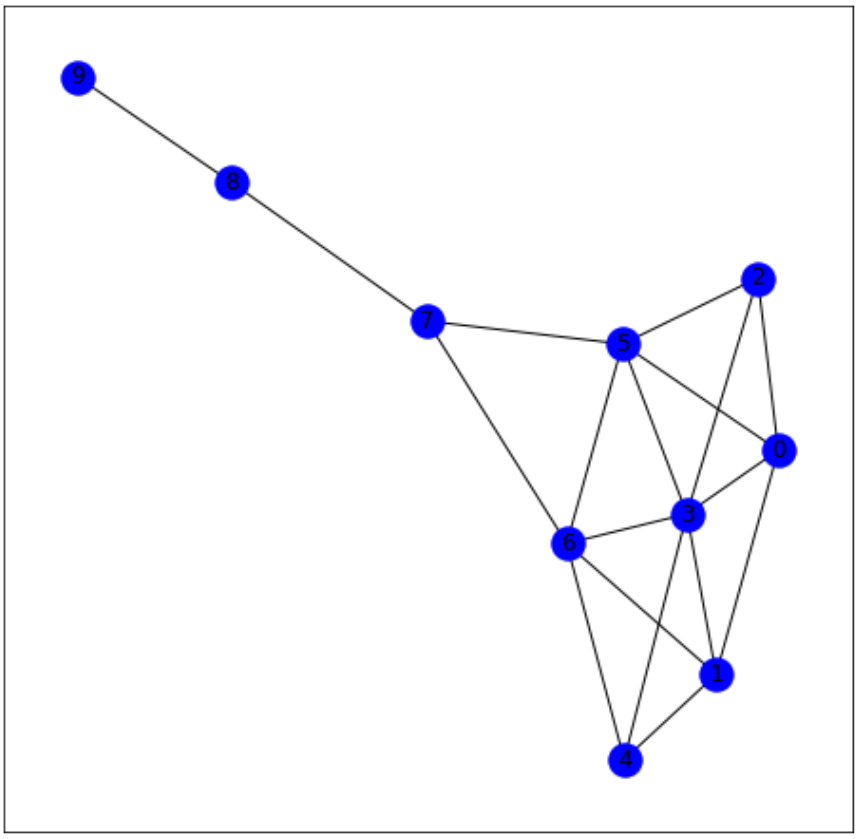
3.2 Closeness Centrality

Degree, in-degree, and out-degree (centrality) consider the direct neighbors of a node, but not indirect neighbors. For instance, if A is connected to B and B is connected to C, the indirect relationship between A and C (through) B might be relevant, whereas it is not taken into consideration in degree, in-degree, and out-degree (centrality). When searching for, for instance, precedents, it might not merely be interesting to look at how many citations it receives, directly or indirectly. Closeness centrality takes such indirect relationships into consideration.

Closeness centrality expresses the average distance between a node and all other nodes it can reach, taking the shortest path between two nodes (called 'geodesic path'). The lower the average distance of one node to all other nodes using the shortest paths, the more central the node is. The details on this are provided below.

The importance of closeness centrality becomes most evident when we have nodes that have a "tail" behind them. They have one direct neighbor, but many indirect neighbors standing in line behind them. This is literally what we see when do the analysis for the kite graph, with the tail of the kite being nodes 7, 8 and 9. Lets first recall the structure of this graph:

```
g_kite = nx.krackhardt_kite_graph()
draw_spring(g_kite)
```



In terms of degree, node 3 is the clear winner in the first position. Node 7 is far behind on the 8th position.

Please note once more that you cannot directly compare different scores, if they have not been normalized (so that the sum of all the scores is 1, and they function as a percentage). Here the direct comparison is not a problem because we are just comparing the rank ordering of the nodes, not the score itself.

```
degree = nx.degree(g_kite)
nodeid = list(dict(degree).keys())
degscore = list(dict(degree).values())
degcentrality = nx.degree_centrality(g_kite)
degcentscore = list(dict(degcentrality).values())
df_reskite1 = pd.DataFrame({"node_id":nodeid,
                      "degree":degscore,
                      "deg_centrality":degcentscore})
df_reskite1.set_index("node_id", inplace=True)
df_reskite1.sort_values('degree', ascending=False)
```

```
         degree  deg_centrality
node_id
```

```
3              6        0.666667
5              5        0.555556
6              5        0.555556
0              4        0.444444
1              4        0.444444
2              3        0.333333
4              3        0.333333
7              3        0.333333
8              2        0.222222
9              1        0.111111
```

However one might think that node 7 is undervalued. It has only 3 direct neighbors, but it actually has many indirect neighbors, including nodes 8 and 9, that can only be reached by passing through 7. Lets see how 7 does when use this second form of centrality score.

```
degree = nx.degree(g_kite)
nodeid = list(dict(degree).keys())
degscore = list(dict(degree).values())
degcentrality = nx.degree_centrality(g_kite)
degcentscore = list(dict(degcentrality).values())
closenesscent = nx.closeness_centrality(g_kite)
closescores = closenesscent.values()


df_reskite2 = pd.DataFrame({"node_id":nodeid,
                      "degree":degscore,
                      "deg_centrality":degcentscore,
                        "deg_norm":score_normalize(degcentscore),
                     "closeness_score":closescores,
                        "closeness_norm": score_normalize(closescores)})
df_reskite2.set_index("node_id", inplace=True)
df_reskite2.sort_values('closeness_norm', ascending=False)
```

```
         degree  deg_centrality  deg_norm  closeness_score  closeness_norm
node_id
5              5        0.555556  0.138889         0.642857        0.121674
6              5        0.555556  0.138889         0.642857        0.121674
3              6        0.666667  0.166667         0.600000        0.113562
7              3        0.333333  0.083333         0.600000        0.113562
0              4        0.444444  0.111111         0.529412        0.100202
1              4        0.444444  0.111111         0.529412        0.100202
2              3        0.333333  0.083333         0.500000        0.094635
4              3        0.333333  0.083333         0.500000        0.094635
8              2        0.222222  0.055556         0.428571        0.081116
9              1        0.111111  0.027778         0.310345        0.058739
```

Things change quite a bit if we look at closeness centrality instead.

Now node 7 moves up from being in the 8th position to being tied for 3rd!

In fact other scores also change. Node 3 is no longer the winner. Under closeness node 3 is penalized for having long paths for nodes 8 and 9. Nodes 5 and 6 obtain better scores. They have less nodes directly linking to them, but they are closer to the other nodes, which is what is relevant for closeness centrality.

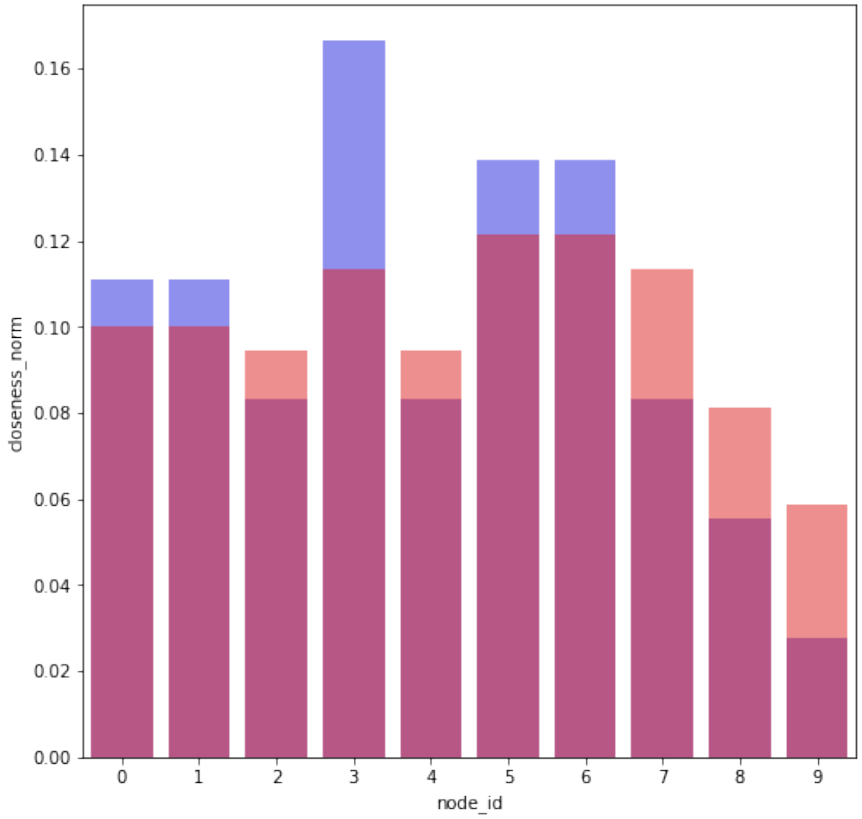
One might intuitively say that node 7 is still underperforming in relation of its real importance. If we cut out node 7, the network becomes disconnected. This is not so if you cut any node from 0 to 6. This will come back when we consider betweenness centrality.

> So which one is better? Degree centrality or closeness? What is crucial to note for now is how different they are. Domain knowledge is needed to see which one matters most.

Here we can compare the relative performance of these two measures directly.

```
plt.figure(figsize=(8,8))
sns.barplot(x = df_reskite2.index, y = df_reskite2.deg_norm, color='blue', alpha␣
↪=0.5)
sns.barplot(x = df_reskite2.index, y = df_reskite2.closeness_norm, color='red',␣
↪alpha = 0.5)
```

```
<AxesSubplot:xlabel='node_id', ylabel='closeness_norm'>
```



Now lets try to unpack the algorithm in more detail.

As noted above, the library can calculate shortest paths for us. Here we get the number of steps.

```
nx.single_source_shortest_path_length(g_kite, source = 8)
```

```
{8: 0, 9: 1, 7: 1, 5: 2, 6: 2, 0: 3, 1: 3, 2: 3, 3: 3, 4: 3}
```

Or if you just want to see the itinerary, and not the number of steps:

```
nx.shortest_path(g_kite, 6, 8)
```

```
[6, 7, 8]
```

The formula to calculate closeness centrality is:

$$\frac{|V|-1}{\sum_{v=1}^{n-1} d(v,u)}$$

In this formula, the number of nodes that can reach a certain specific node (called u here) minus 1, is divided by the sum of minimum path lengths between each of those nodes v and the target node (node u).

The top part of the formula expresses the number (cardinality) of nodes that can reach node u (minus 1), whereas d in the bottom part of the formula refers to the length of the shortest path between nodes v and u. That is the smallest number of steps that you have to take from node v to reach node u.

The number of steps for every v paired with u are added up and then that is used to divide the total number of nodes v that can reach u.

(Note that sums are normally expressed by a sigma notation rather than by a+b+c+d. The Σ means 'sum', the subscript (the a at the bottom) gives you the element where the sum begins, and the superscript (the letter at the top) the element where it ends.)

$$\sum_{a}^{d}$$

A) Closeness with weights (distances) We previously explained that edges can have weights and that networks can be fully connected. A network of document similarity has both features. Imagine we want to measure the semantic similarity between two cases, for instance because we want to compare how similar decisions are in terms of their facts, the reasoning, or simply on the document level.

What would be the expected result of running a centrality measure, such as closeness centrality, on this network without specifying any weight information? The result would be that all nodes are equally - maximally - close to each other. They would all be connected (degree=1) and equally close to each other (closeness=1), considering that all the cases have a similarity score: even though pairs of cases have different similarity scores, they all have *a* similarity score.

```
res = nx.closeness_centrality(g_docs)
res_view = [ (v,k) for k,v in res.items() ]
res_view.sort(reverse=True)
for v,k in res_view:
    print(f"for node {k} closeness score is {np.round(v,2)}")
```

```
for node 8 closeness score is 1.0
for node 7 closeness score is 1.0
for node 6 closeness score is 1.0
for node 5 closeness score is 1.0
for node 4 closeness score is 1.0
for node 3 closeness score is 1.0
for node 2 closeness score is 1.0
for node 1 closeness score is 1.0
for node 0 closeness score is 1.0
```

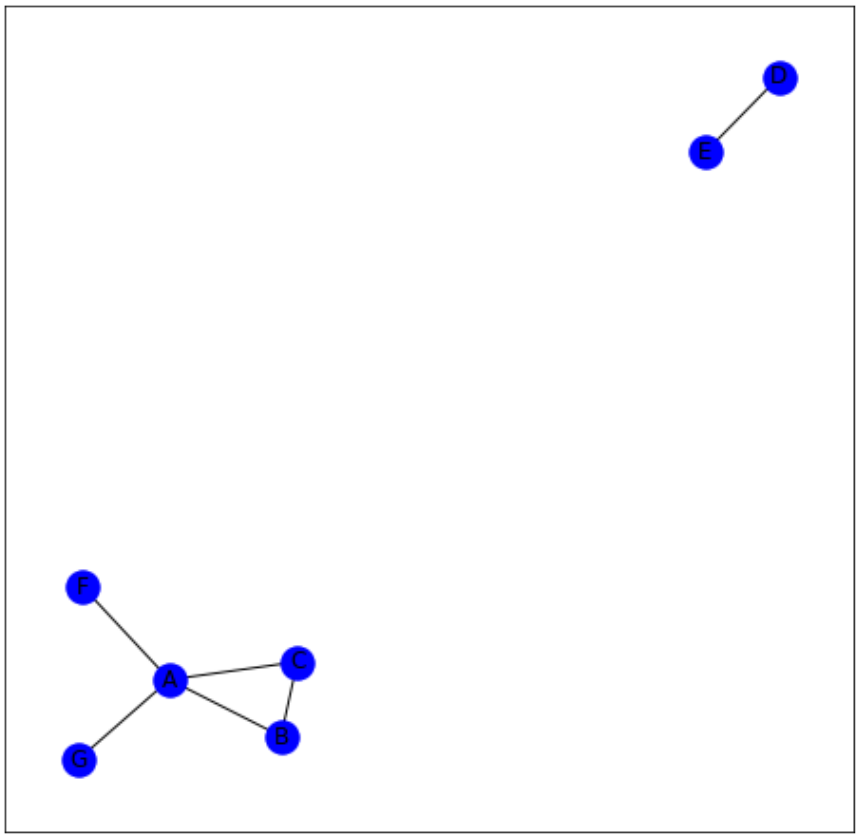
To avoid this uninformative result, it is possible to adjust the closeness metric considering the weights, with the weights indicating document similarity (the higher the weight, the more similar the document).

```
g_docs = load_graph_from_json("data/g_docs.json")
res = nx.closeness_centrality(g_docs, distance='weight')
res_view = [ (v,k) for k,v in res.items() ]
res_view.sort(reverse=True)
for v,k in res_view:
    print(f"for node {k} closeness score is {np.round(v,2)}")
```

```
for node 8 closeness score is 2.8
for node 1 closeness score is 2.72
for node 2 closeness score is 2.7
for node 7 closeness score is 2.57
for node 3 closeness score is 2.52
for node 0 closeness score is 2.5
for node 5 closeness score is 1.98
for node 6 closeness score is 1.87
for node 4 closeness score is 1.23
```

It is good to know that closeness centrality takes into consideration how well-connected a node is (the numerator of the formula increases when more nodes can reach the node of interest). Consider the following example, where nodes D and E have a single very short path between them. This path is very short, but it is penalized by the very small numerator, as D and E can only be reached by one other node.

```
g_disconnected2 = nx.Graph()
g_disconnected2.add_nodes_from(['A','B','C','D','E','F','G'])
g_disconnected2.add_edges_from([('A','B'),('B','C'), ('C','A'),␣
 ↪('A','F'),('A','G'),('D','E')])
draw_spring(g_disconnected2)
```




```
nx.closeness_centrality(g_disconnected2)
```

```
{'A': 0.6666666666666666,
 'B': 0.4444444444444444,
 'C': 0.4444444444444444,
 'D': 0.16666666666666666,
 'E': 0.16666666666666666,
 'F': 0.38095238095238093,
 'G': 0.38095238095238093}
```

3.3 Eigenvector Centrality

Another approach to determine the centrality of a node is to consider how likely it is to reach that node if one would randomly 'walk' through the network. If a node would be reached more frequently than other nodes, this could signal that the node is more central and consequently more relevant.

Eigenvector centrality is based on this 'random walker' idea. It simulates a random walker moving from node to node, each time with equal probability of going to the adjacent node. The idea is that if the walk goes on for very long, the probability of seeing the walker at a particular node will

stabilize, giving you the Eigenvector Centrality value.

To illustrate, we calculate the Eigenvector Centrality scores for the train stations.

```
res = nx.eigenvector_centrality(g_trains)
resdf = pd.DataFrame({"station":res.keys(), "score":res.values()})
resdf.set_index("station")
resdf.sort_values('score', ascending=False)
```

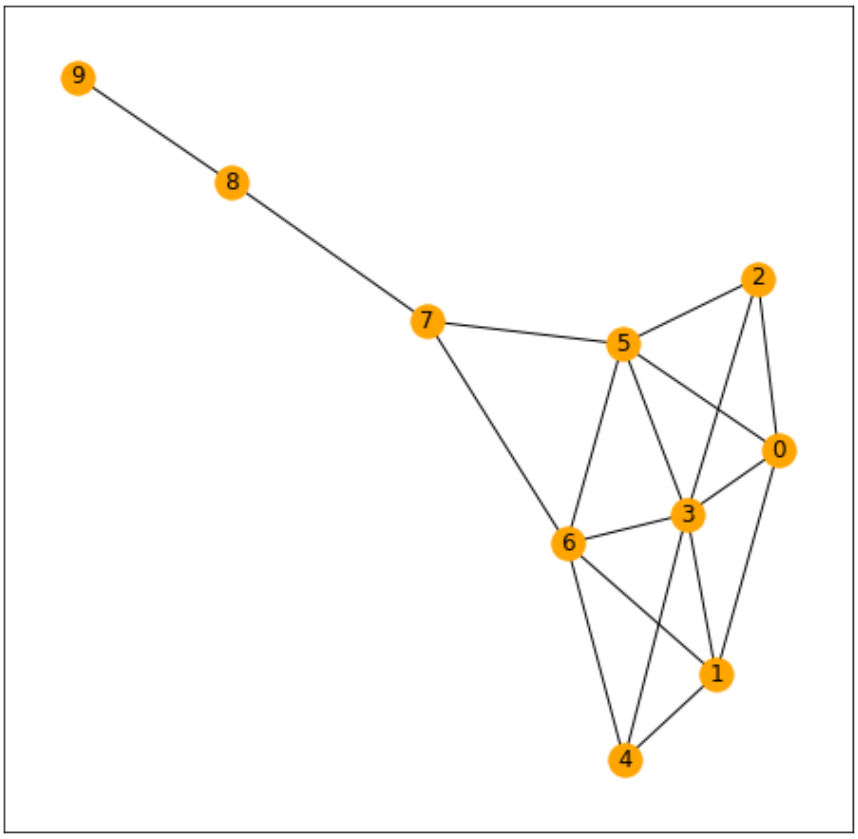
	station	score
1	Utrecht	0.569051
0	Amsterdam	0.389746
9	Groningen	0.267149
15	Zwolle	0.267149
19	Schiphol	0.264338
4	Eindhoven	0.254248
14	The Hague	0.246852
6	Venlo	0.216722
13	Rotterdam	0.174007
7	Nijmegen	0.171231
2	Gouda	0.161922
11	Putten	0.124004
12	Gelderlmalsen	0.124004
10	Groningen	0.124004
3	Weert	0.058312
16	Tilburg	0.058312
5	Roermond	0.013341
17	Roosendaal	0.013341
8	Maastricht	0.002907
18	Middelburg	0.002907

Now lets remember that degree centrality favors the number of direct neighbors one has, closeness centrality takes into account indirect networks, including thus nodes that are in a "tail" formation behind a node. Eigenvector centrality has an iterative nature (where is the random walker likely to be after many walks), and what this does, is that it takes into account the relative importance of nodes. A node gets a high eigenvector centrality score if it is linked to other nodes which also have a high eigenvector centrality score. So the intuition is that here being "famous" is defined in terms of having many friends who are also "famous".

Lets go back to the kite and remember that: * Degree centrality prefers node 3, and puts node 7 far on the back. * Closeness prefer both 5 and 6, and puts node 7 in 4th place Now we will see that * Eigenvector centrality prefers node 3, and also puts node 7 far on the back.

Note why this might be so. Node 3 not only has high degree, but it is also close to nodes 5, 6, 0 and 1, which also have high degree. By contrast node 7 is "friends" with 5 and 6, but also with the unpopular 8, so it can rank as high.

```
draw_spring(g_kite, node_color="orange")
```



We can see the scores here.

```
degree = nx.degree(g_kite)
nodeid = list(dict(degree).keys())
degscore = list(dict(degree).values())
degcentrality = nx.degree_centrality(g_kite)
degcentscore = list(dict(degcentrality).values())
closenesscent = nx.closeness_centrality(g_kite)
closescores = closenesscent.values()
eigcentrality = nx.eigenvector_centrality(g_kite)
eigscore = list(dict(eigcentrality).values())
reskite3 = pd.DataFrame({"node_id":nodeid,
                         "degree":degscore,
                         "deg_centrality":degcentscore,
                            "closeness_cent": closescores,
                           "eig_centrality":eigscore})
reskite3.set_index("node_id", inplace=True)
reskite3.sort_values('eig_centrality', ascending=False)
```

node_id	degree	deg_centrality	closeness_cent	eig_centrality
3	6	0.666667	0.600000	0.481020
5	5	0.555556	0.642857	0.397691
6	5	0.555556	0.642857	0.397691
0	4	0.444444	0.529412	0.352209
1	4	0.444444	0.529412	0.352209
2	3	0.333333	0.500000	0.285835
4	3	0.333333	0.500000	0.285835
7	3	0.333333	0.600000	0.195862
8	2	0.222222	0.428571	0.048075
9	1	0.111111	0.310345	0.011164

The mathematics of Eigenvector Centrality are complicated, in particular for non-technical persons. We are not going to present it in full, but we will, in the next example, sketch an idea of how Eigenvector Centrality works and some ways of arriving at it.

In its simplest form, Eigenvector Centrality is what you get by repeatedly multiplying:

- an **adjacency matrix** by
- a **vector of probabilities** associated with each node.

Let us use a small network to illustrate. Imagine we have four nodes "a", "b", "c" and "d", and we are making a network of their linkages. We begin representing this network as a 4×4 adjacency matrix "A" (it is customary to represent matrices with capital letters). In this matrix a 1 indicates there is a connection between a node in the rows and another in the columns, and a 0 means that there is no connection. If one picks node "a" in the rows, we see that it connects to "b" and "d" in the columns. If we read the row of node "b", we see that it connects with "a" and "d" in the columns. And so on.

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

For the vector of initial probabilities "v" we can just assume that one is equally likely to be in each of the four rows of the matrix.

$$v = (0.25, 0.25, 0.25, 0.25)$$

The basic idea is that if one multiplies "A" and "v" one will "apply" the probabilities of the contingency matrix to the vector, or "scale" the vector by the probability matrix. If one does this multiple times, thousands of times, the values will "converge", which means they will not change much from multiplication to multiplication.

$$v := A \times v$$

Afterwards the vector can be "normalized" by dividing it by its norm:

$$\frac{v}{||v||}$$

The result represents the eventual stable position of many repeated random walks.

Why is this called Eigenvector Centrality? Because a property of the adjacency matrix, namely, the Eigenvector associated with its highest Eigenvalue, will give the same results as the method presented above.

Let us put the procedure in practice. We first create the adjacency matrix of the kite graph.

```
A = nx.adjacency_matrix(g_kite)
A.todense()
```

```
matrix([[0, 1, 1, 1, 0, 1, 0, 0, 0, 0],
        [1, 0, 0, 1, 1, 0, 1, 0, 0, 0],
        [1, 0, 0, 1, 0, 1, 0, 0, 0, 0],
        [1, 1, 1, 0, 1, 1, 1, 0, 0, 0],
        [0, 1, 0, 1, 0, 0, 1, 0, 0, 0],
        [1, 0, 1, 1, 0, 0, 1, 1, 0, 0],
        [0, 1, 0, 1, 1, 1, 0, 1, 0, 0],
        [0, 0, 0, 0, 0, 1, 1, 0, 1, 0],
        [0, 0, 0, 0, 0, 0, 0, 1, 0, 1],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]], dtype=int32)
```

The next step is to have an initial vector of probabilities. Each element of this vector should be 1/10 or 0.1. Remember that probabilities have to sum to 1, and if it is equally probable that one may be in each of the 10 nodes of the kite graph, then we have a 0.1 probability for each.

```
v = (np.ones(10)/10)
v
```

```
array([0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1])
```

Finally we repeatedly multiply the adjacency matrix by the vector of probabilities and normalize the results. We will just do it 100 times. In more complicated scenarios, more repetitions might be required to do the trick.

After the multiplications are done, we normalize the results by dividing the results vector by its norm.

```
for i in range(0,100):
    v = A@v
mynorm = np.linalg.norm(v)
res = v / mynorm
res
```

```
array([0.3522094 , 0.3522094 , 0.28583499, 0.48102086, 0.28583499,
       0.39769064, 0.39769064, 0.19586058, 0.04807349, 0.01116326])
```

NetworkX gives the results directly:

```
nx.eigenvector_centrality(g_kite)
```

```
{0: 0.3522089813920359,
 1: 0.3522089813920358,
 2: 0.28583473531632403,
 3: 0.48102048812210046,
 4: 0.28583473531632403,
 5: 0.3976910106255469,
 6: 0.39769101062554685,
 7: 0.19586185175360382,
 8: 0.048074775014202924,
 9: 0.011164058575824235}
```

Finally let us approach it from the perspective of eigenvalues and eigenvectors.

Eigenvalues and eigenvectors are a property of certain matrices, which are complicated to get by hand, but which numpy can provide on command.

Eigenvectors are just vectors, that when multiplied by the matrix ($matrix \times vector$), scale, but do not rotate it.

Every eigenvector will have an eigenvalue -just a number- associated with it, which records how much the vector is scaled. Here we are interested in the eigenvector that is associated with the largest eigenvalue

The largest eigenvalue here is 4.3. This is in index position 0 of the numpy array (arrays like a python list, start from zero).

```
vals, vecs = np.linalg.eig(A.todense())
vals
```

```
array([ 4.30640379,  1.60639741,  1.41421356,  0.64036468, -0.40434195,
       -0.816375  , -1.41421356, -1.46406328, -2.        , -1.86838565])
```

If we see the eigenvector that is in position 0 (corresponding to the largest eigenvalue that we recall was in index position 0), we again get the result we are looking for.

```
vecs[:,0]
```

```
matrix([[0.3522094 ],
        [0.3522094 ],
        [0.28583499],
        [0.48102086],
        [0.28583499],
        [0.39769064],
        [0.39769064],
        [0.19586058],
        [0.04807349],
```

```
        [0.01116326]])
```

Note what is going on here code wise. `vecs` gives us all the eigenvectors.

```
vecs
```

```
matrix([[ 3.52209397e-01,  1.89445487e-01, -3.53553391e-01,
          2.50475739e-01,  5.05030447e-01,  7.62482192e-02,
          3.53553391e-01, -1.27468135e-01,  5.00000000e-01,
         -1.42542367e-02],
        [ 3.52209397e-01,  1.89445487e-01,  3.53553391e-01,
          2.50475739e-01,  5.05030447e-01,  7.62482192e-02,
         -3.53553391e-01, -1.27468135e-01, -5.00000000e-01,
         -1.42542367e-02],
        [ 2.85834991e-01,  1.16760682e-01, -5.00000000e-01,
          9.77805840e-02, -3.42822530e-01, -3.38990557e-01,
         -5.00000000e-01, -4.02146483e-01, -2.78838203e-16,
         -3.06688753e-02],
        [ 4.81020858e-01,  1.68769387e-01, -8.45844030e-16,
          1.21495199e-01, -2.55950827e-01, -1.05180509e-01,
         -3.07079383e-15,  6.96430792e-01,  7.80943502e-16,
          4.04703757e-01],
        [ 2.85834991e-01,  1.16760682e-01,  5.00000000e-01,
          9.77805840e-02, -3.42822530e-01, -3.38990557e-01,
          5.00000000e-01, -4.02146483e-01, -6.60595327e-16,
         -3.06688753e-02],
        [ 3.97690636e-01, -1.70650816e-01, -3.53553391e-01,
         -3.09355706e-01, -1.10462089e-01,  3.05675707e-01,
          3.53553391e-01,  1.98052427e-02, -5.00000000e-01,
         -3.33148234e-01],
        [ 3.97690636e-01, -1.70650816e-01,  3.53553391e-01,
         -3.09355706e-01, -1.10462089e-01,  3.05675707e-01,
         -3.53553391e-01,  1.98052427e-02,  5.00000000e-01,
         -3.33148234e-01],
        [ 1.95860583e-01, -5.78457769e-01,  2.47118996e-15,
         -3.58496284e-01,  2.48869455e-01, -1.87298866e-01,
          9.27452560e-16, -2.15617545e-01,  1.33498006e-16,
          5.95816968e-01],
        [ 4.80734850e-02, -5.87931432e-01,  3.69155892e-15,
          3.89143053e-01,  1.20295815e-01, -4.58445302e-01,
         -1.04245797e-15,  2.76067245e-01,  8.43463621e-17,
         -4.46919404e-01],
        [ 1.11632553e-02, -3.65993762e-01,  2.26512264e-15,
          6.07689751e-01, -2.97510100e-01,  5.61562150e-01,
          8.75869888e-16, -1.88562372e-01,  4.62805930e-17,
          2.39200833e-01]])
```

Each of the eigenvectors (associated with every eigenvalue) is in a column. Using `vecs[:,0]` tell us

give me all the rows, under column zero.

```
vecs[:,0]
```

```
matrix([[0.3522094 ],
        [0.3522094 ],
        [0.28583499],
        [0.48102086],
        [0.28583499],
        [0.39769064],
        [0.39769064],
        [0.19586058],
        [0.04807349],
        [0.01116326]])
```

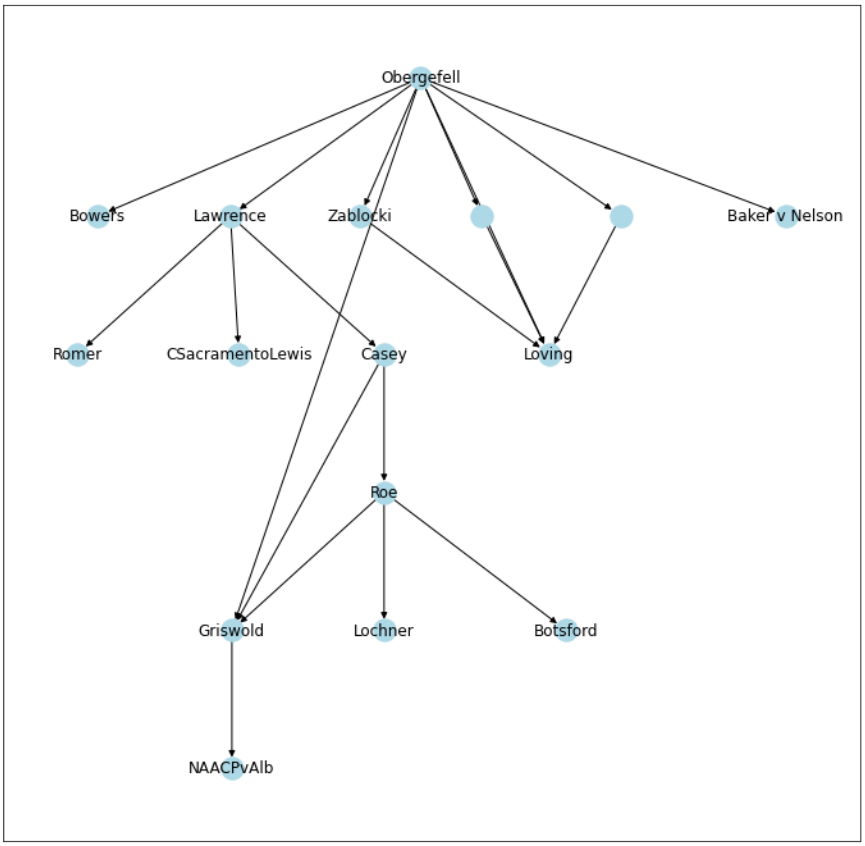
3.4 PageRank

PageRank is a variant of the Eigenvector centrality which uses in-degree centrality (backlinks) as a relative measure of influence and thus is applied to directed networks. The PageRank algorithm is based on the idea that important pages are linked to more often by other pages. Intuitively, the rank of a page corresponds to the likelihood of a user ending up on that page by randomly clicking from link to link. PageRank used to form the basis for the Google search engine algorithm.

Because pagerank simulates movement from one website to another through its links, it works only on directed networks. Here we can see how it might work in a simplified citation network of the American case Obergefell v. Hodges (the names of cases Turner v. Safley and Windsor v. United States have been omitted to de-clutter the plot)

```
g_ob = load_graph_from_json("data/obergefell.json")
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(12,12))
pos = graphviz_layout(g_ob, 'dot')
nx.draw_networkx_nodes(g_ob, pos=pos, node_color='lightblue', ax=ax)
nx.draw_networkx_edges(g_ob, pos=pos, ax=ax)
nx.draw_networkx_labels(g_ob, pos=pos, labels= {"Obergefell_v_Hodges":␣
↪"Obergefell",
                                                "Loving_v_Virginia":"Loving",
                                                "NAACP_v_State_of_Alabama":
↪"NAACPvAlb",
                                               ␣
↪"Union_Pacific_Railroad_v_Botsford":"Botsford",
                                               "Lochner_v_New_York":"Lochner",
                                               "Griswold_v_Connecticut":
↪"Griswold",
                                               "Roe_v_Wade":"Roe",
                                               "PPvCasey":"Casey",
                                               "Lawrence_v_Texas":"Lawrence",
```

```
                                                    "Zablocki_v_Redhail":"Zablocki",
                                                    "Baker_v_Nelson":"Baker v Nelson",
                                                    "County_of_Sacramento_vs._Lewis":
↪"CSacramentoLewis",
                                                    "Romer_v_Evans":"Romer",
                                                    "Bowers_v_Hardwick":"Bowers"},␣
↪ax=ax);
```




```
nx.pagerank(g_ob, alpha=0.90)
```

```
{'Obergefell_v_Hodges': 0.0391864140766086,
 'Lawrence_v_Texas': 0.043594930939948255,
 'Bowers_v_Hardwick': 0.043594930939948255,
 'Zablocki_v_Redhail': 0.043594930939948255,
 'Turner_v_Safley': 0.043594930939948255,
 'Windsor_v_United_States': 0.043594930939948255,
 'PPvCasey': 0.05226493501064501,
 'Roe_v_Wade': 0.06270557860630709,
 'Griswold_v_Connecticut': 0.08592586114620163,
 'Lochner_v_New_York': 0.05799817975316349,
```

```
'NAACP_v_State_of_Alabama': 0.116519776646655,
'Union_Pacific_Railroad_v_Botsford': 0.05799817975316349,
'Loving_v_Virginia': 0.16130161934627593,
'Baker_v_Nelson': 0.043594930939948255,
'County_of_Sacramento_vs._Lewis': 0.05226493501064501,
'Romer_v_Evans': 0.05226493501064501}
```

PageRank incorporates a "teleportation" factor: As a random walker moves through the network, it may get stuck on a node or nodes that has only incoming links to it, and no outgoing links. For example, Loving v. Virginia or Lochner v. New York. The teleportation factors implies a random chance of teleporting anywhere else in the network and thus avoiding such dead ends.

To correct for this, the standard pagerank model gives a small random percentage of "teleporting" to any node in the network. The default value for this in NetworkX is 0.15.

The details are as follows:

The first thing PageRank needs, is to transform the adjacency matrix into a transition matrix. While an adjacency matrix will record a 1 if there is a connection between a node in the rows and a node in the column, the transition matrix will record the probability of jumping from a node in the rows to another one in the columns in such a way that the probabilities all sum to 1.

Imagine once more we have four metro stations A, B, C, D, that are connected as follows:

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

If we are in station A (first row), we can jump to station B or D. If we are in station B (second row) we can jump to station A and D. Granting equal probability of making that jump, we now weight the connection in terms of probabilities as in this manner:

$$\begin{pmatrix} 0 & 0.5 & 0 & 0.5 \\ 0.5 & 0 & 0 & 0.5 \\ \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} \\ 0 & 0.5 & 0.5 & 0 \end{pmatrix}$$

Note that each row will sum to 1 as per the rules of probability.

Just like with Eigenvector centrality, the next thing needed is a vector that reflects the probability of being in each of the nodes of the network. There are four nodes, so a vector with equal probability will be a vector of 0.25s.

$$v = (0.25, 0.25, 0.25, 0.25)$$

In principle what needs to be done next is to multiply the vector of probabilities by the transition matrix many times.

$$v := T^T \times v$$

However, if this algorithm is used, nodes that are sinks (have no outgoing links) or which form a closed circuit will trap all the pagerank scores. So something needs to be changed. What happens is that the transition matrix T is modified by adding a teleportation factor Beta to it, so that for every node there is a 1-Beta chance of getting out. It is standard to put Beta at 0.85.

In NetworkX the transition matrix so modified is called the google matrix, and so we are going to denote it with the letter G.

$$G = T * Beta + [(1 - Beta)/n]_{nxn}$$

Now we can go ahead and repeatedly multiply G and the vector of probabilities.

$$v := G^T \times v$$

Just like with Eigenvector Centrality, eventually the values of v will not change much from multiplication to multiplication, leading us to our score (unnormalized).

Pagerank is presented as a percentage, so then we normalize this value by dividing it for its sum:

$$pagerank = \frac{v}{\sum v}$$

This is how we can get the scores using a for loop.

```
A = nx.google_matrix(g_ob).T
v = np.ones(16)/16

for i in range(0, 100):
    v = A @ v.reshape(16,-1)
    vnorm = np.linalg.norm(v)
    v = v/vnorm

result = np.ravel(v/sum(v))
```

```
official_pr = nx.pagerank(g_ob)

pd.DataFrame({"nodes":official_pr.keys(), "official": official_pr.values(),␣
 ↪"unofficial":result})
```

	nodes	official	unofficial
0	Obergefell_v_Hodges	0.040254	0.040254
1	Lawrence_v_Texas	0.044531	0.044531
2	Bowers_v_Hardwick	0.044531	0.044531
3	Zablocki_v_Redhail	0.044531	0.044531
4	Turner_v_Safley	0.044531	0.044531
5	Windsor_v_United_States	0.044531	0.044531

```
6                           PPvCasey  0.052871    0.052871
7                         Roe_v_Wade  0.062724    0.062724
8              Griswold_v_Connecticut  0.084773    0.084773
9                  Lochner_v_New_York  0.058026    0.058026
10           NAACP_v_State_of_Alabama  0.112311    0.112311
11  Union_Pacific_Railroad_v_Botsford  0.058026    0.058026
12                 Loving_v_Virginia  0.158085    0.158085
13                    Baker_v_Nelson  0.044531    0.044531
14     County_of_Sacramento_vs._Lewis  0.052871    0.052871
15                      Romer_v_Evans  0.052871    0.052871
```

And as expected, the eigenvalues associated with the largest eigenvector give us the same result.

```
A = nx.google_matrix(g_ob).T
vals, vecs = np.linalg.eig(A)
argmax = np.argmax(vals)
eigresult = np.ravel(vecs[:,argmax])
eigresult = eigresult/np.sum(eigresult)
```

```
pd.DataFrame({"nodes":official_pr.keys(), "official": official_pr.values(),␣
↪"unofficial":result, "eigen":eigresult})
```

```
                            nodes  official  unofficial  \
0                 Obergefell_v_Hodges  0.040254    0.040254
1                    Lawrence_v_Texas  0.044531    0.044531
2                  Bowers_v_Hardwick  0.044531    0.044531
3                 Zablocki_v_Redhail  0.044531    0.044531
4                    Turner_v_Safley  0.044531    0.044531
5             Windsor_v_United_States  0.044531    0.044531
6                           PPvCasey  0.052871    0.052871
7                         Roe_v_Wade  0.062724    0.062724
8              Griswold_v_Connecticut  0.084773    0.084773
9                  Lochner_v_New_York  0.058026    0.058026
10           NAACP_v_State_of_Alabama  0.112311    0.112311
11  Union_Pacific_Railroad_v_Botsford  0.058026    0.058026
12                 Loving_v_Virginia  0.158085    0.158085
13                    Baker_v_Nelson  0.044531    0.044531
14     County_of_Sacramento_vs._Lewis  0.052871    0.052871
15                      Romer_v_Evans  0.052871    0.052871

                eigen
0   0.040254+0.000000j
1   0.044531+0.000000j
2   0.044531+0.000000j
3   0.044531+0.000000j
4   0.044531+0.000000j
5   0.044531+0.000000j
```

```
6   0.052871+0.000000j
7   0.062724+0.000000j
8   0.084773+0.000000j
9   0.058026+0.000000j
10  0.112311+0.000000j
11  0.058026+0.000000j
12  0.158085+0.000000j
13  0.044531+0.000000j
14  0.052871+0.000000j
15  0.052871+0.000000j
```

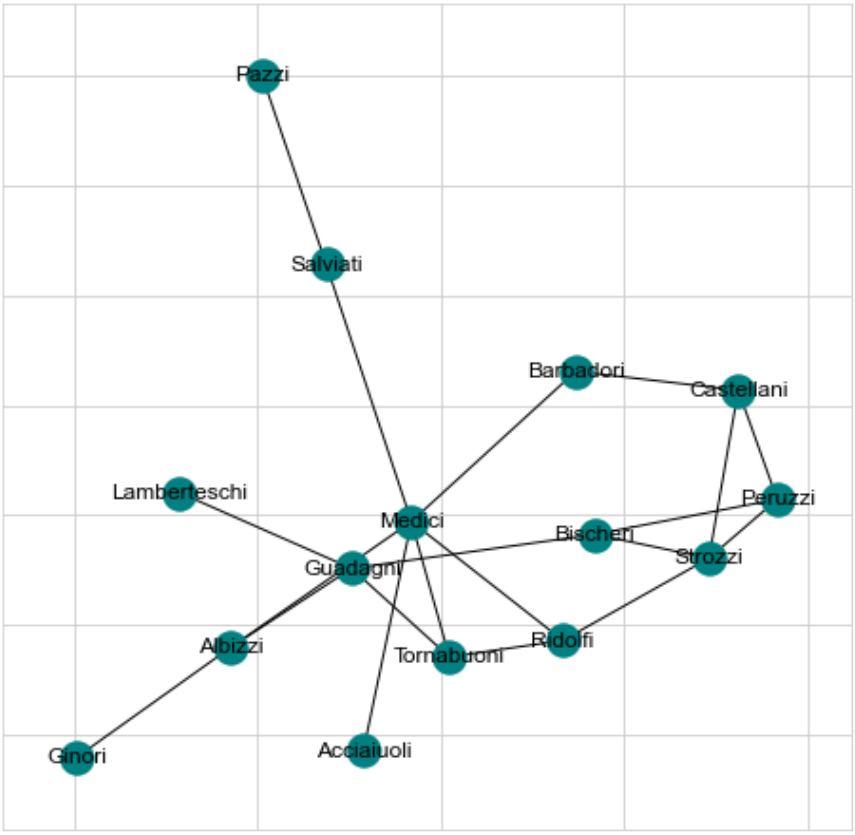
3.5 Betweenness Centrality

So far we have been thinking of centrality in relation to the number of neighbors a node has. 1. Degree centrality simply focuses on the number of direct neighbors a node has. 2. Closeness centrality takes into account not just direct neighbors, but also indirectly linked nodes. 3. Eigenvector centrality and Pagerank consider the connectedness or "fame" of the neighbors.

Now we will introduce a different focus. We will consider as most important not just nodes that have a lot of neighbors, but nodes that are "mediators", or "brokers" in the network. These are nodes that hold the network together. If they are deleted, this will cause severe disruption in the network, possibly disconnecting parts of the network. Betweenness centrality considers this broker status. In the law, for inst.ance case law, betweenness centrality can be used to measure which court decisions have played a pivotal role in the development of a certain legal topic or area of the law, for instance consumer protection, intent in criminal law, data protection law, or negligence.

To illustrate how betweenness centrality can be determined or explained, we start with the example of actual political brokerage. The "Florentine Families" graph is a study of marriage alliances between families in Renaissance Florence. However, the visualization of the graph does not immediately allow us to see the dominance of the Medici...

```
draw_spring(g_florentine, node_color="teal")
```



Conventional centrality measures will show that the Medici have a stronger political situation, but they might understate their dominance. The closeness score for Medici is 0.0899, but Strozzi, Barbadori, Ridolfi, Tornabuouni, Albizzi and Guadagni are not far behind. Once we factor in betweenness, the scores for the aforementioned families drop significantly and we get far better impression of the dominance of the Medici.

```
closeness_arr = np.array([x for x in nx.closeness_centrality(g_florentine).
 ↪values()])
closeness_sum = np.sum(closeness_arr)
closeness_norm = closeness_arr/closeness_sum
betweenness_arr = np.array([x for x in nx.betweenness_centrality(g_florentine).
 ↪values()])
betweenness_sum = np.sum(betweenness_arr)
betweenness_norm = betweenness_arr/betweenness_sum


df_medici = pd.DataFrame({'families':nx.closeness_centrality(g_florentine).
 ↪keys(),
                          'closeness_norm': closeness_norm,
                         'betweenness_norm': betweenness_norm})
```

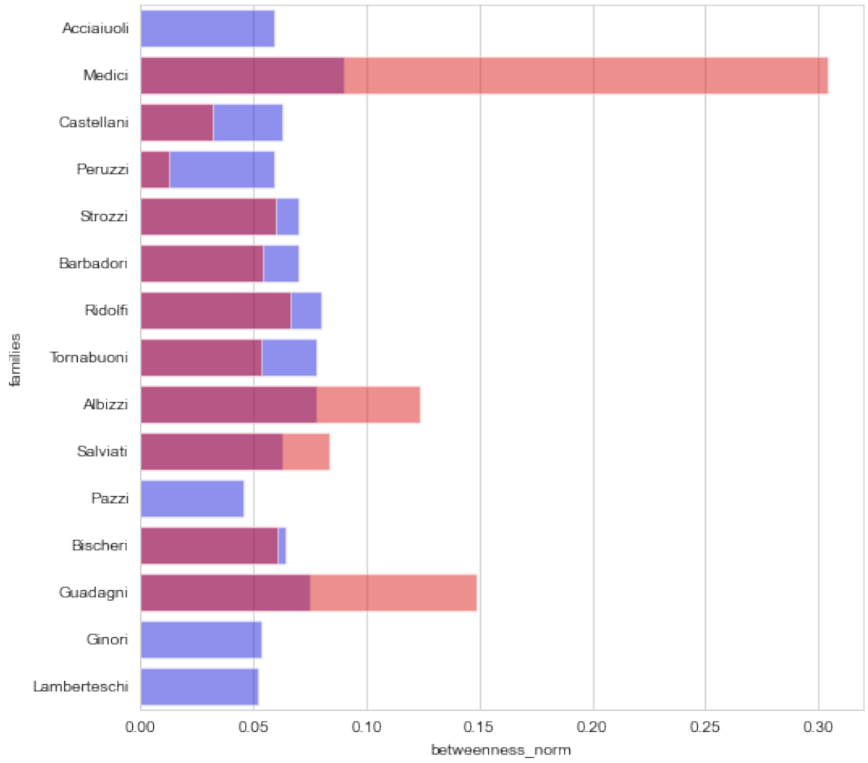
```
df_medici
```

	families	closeness_norm	betweenness_norm
0	Acciaiuoli	0.0592	0.0000
1	Medici	0.0899	0.3045
2	Castellani	0.0625	0.0321
3	Peruzzi	0.0592	0.0128
4	Strozzi	0.0703	0.0598
5	Barbadori	0.0703	0.0545
6	Ridolfi	0.0803	0.0662
7	Tornabuoni	0.0775	0.0534
8	Albizzi	0.0775	0.1239
9	Salviati	0.0625	0.0833
10	Pazzi	0.0459	0.0000
11	Bischeri	0.0642	0.0609
12	Guadagni	0.0749	0.1485
13	Ginori	0.0535	0.0000
14	Lamberteschi	0.0523	0.0000

Here we can see the massive contrast. The blue bars highlight the closeness centrality, and the red bars betweenness centrality. (If a blue bar exceeds a red bar, this indicates the extent to which the closeness centrality score is higher than the betweenness centrality score.)

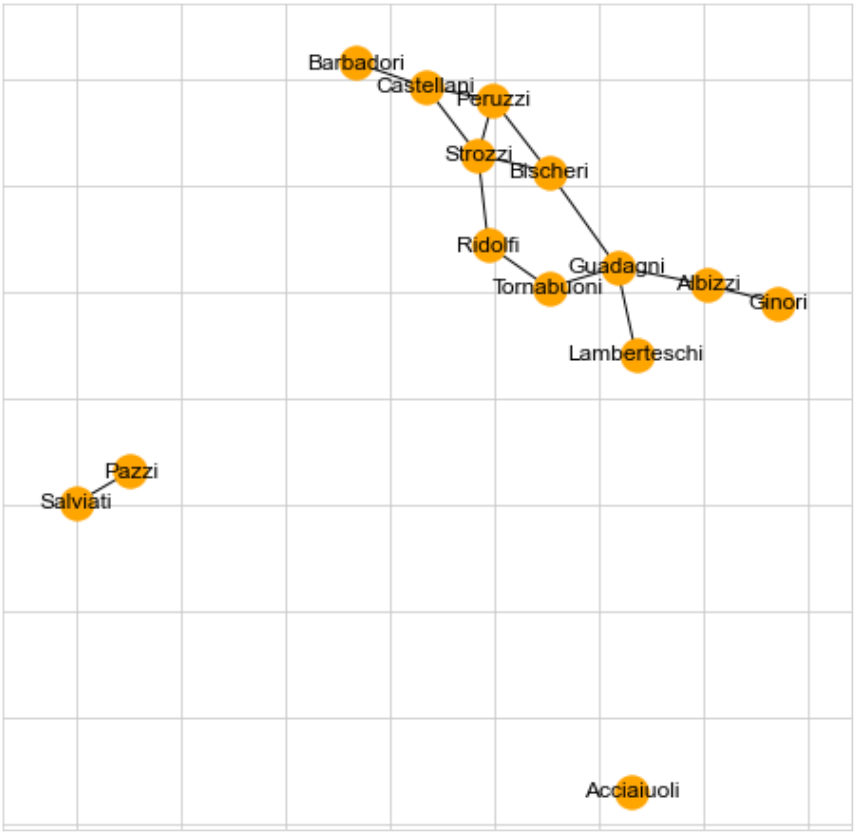
```
plt.figure(figsize=(8,8))
sns.barplot(y = df_medici.families, x = df_medici.closeness_norm, color='blue',␣
↪alpha =0.5)
sns.barplot(y = df_medici.families, x = df_medici.betweenness_norm, color='red',␣
↪alpha = 0.5)
```

```
<AxesSubplot:xlabel='betweenness_norm', ylabel='families'>
```



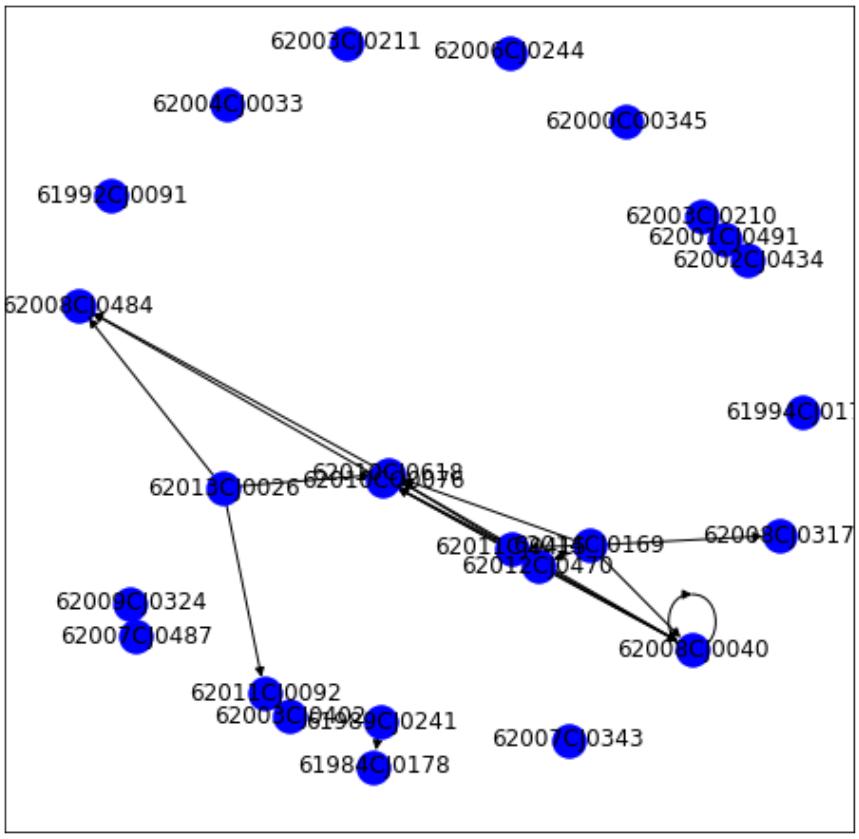
Finally, let's see what happens when we remove the Medici family from the graph. As we can see, the graph suddenly becomes disconnected.

```
g_subflorentine = g_florentine.subgraph([x for x in g_florentine.nodes if not␣
↪x=="Medici"])
draw_spring(g_subflorentine, node_color="orange")
```



We can perform a similar analysis for the subgraph of our CJEU case law network. We start with a comparison between the closeness and betweenness centrality scores of the cases. We first display the network.

```
g_consprot2 = load_graph_from_json("data/g_consprot2.json")
draw_spring(g_consprot2)
```



Like in the Medici family example, we compare the closeness and betweenness centrality scores...

```
closeness_arr = np.array([x for x in nx.closeness_centrality(g_consprot2).
 ↪values()])
closeness_sum = np.sum(closeness_arr)
closeness_norm = closeness_arr/closeness_sum
betweenness_arr = np.array([x for x in nx.betweenness_centrality(g_consprot2).
 ↪values()])
betweenness_sum = np.sum(betweenness_arr)
betweenness_norm = betweenness_arr/betweenness_sum


df_consprot = pd.DataFrame({'cases':nx.closeness_centrality(g_consprot2).keys(),
                        'closeness_norm': closeness_norm,
                       'betweenness_norm': betweenness_norm})
df_consprot
```

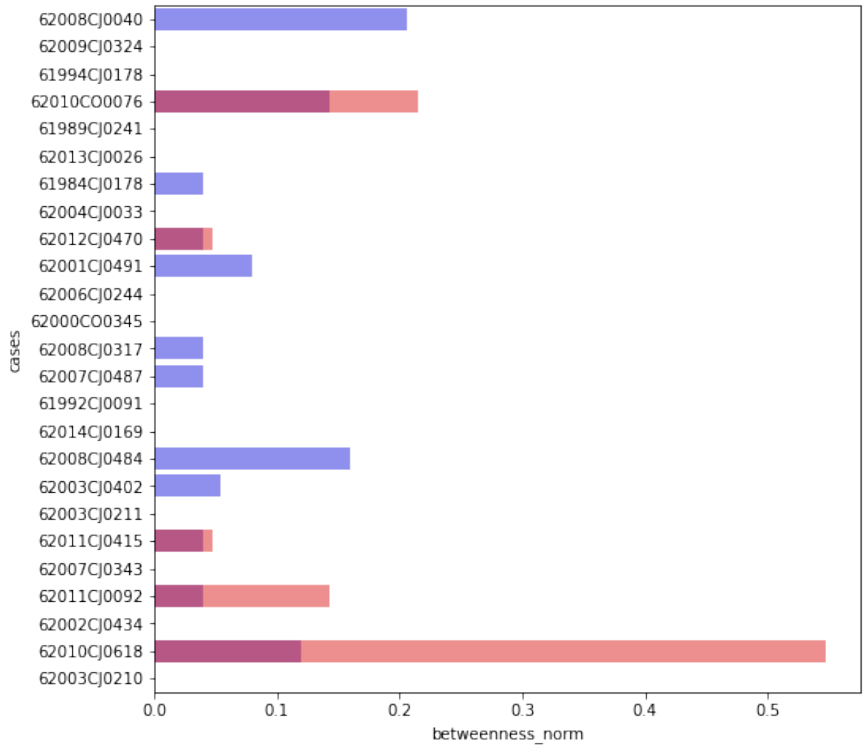
	cases	closeness_norm	betweenness_norm
0	62008CJ0040	0.205323	0.000000
1	62009CJ0324	0.000000	0.000000

2	61994CJ0178	0.000000	0.000000
3	62010CO0076	0.142586	0.214286
4	61989CJ0241	0.000000	0.000000
5	62013CJ0026	0.000000	0.000000
6	61984CJ0178	0.039924	0.000000
7	62004CJ0033	0.000000	0.000000
8	62012CJ0470	0.039924	0.047619
9	62001CJ0491	0.079848	0.000000
10	62006CJ0244	0.000000	0.000000
11	62000CO0345	0.000000	0.000000
12	62008CJ0317	0.039924	0.000000
13	62007CJ0487	0.039924	0.000000
14	61992CJ0091	0.000000	0.000000
15	62014CJ0169	0.000000	0.000000
16	62008CJ0484	0.159696	0.000000
17	62003CJ0402	0.053232	0.000000
18	62003CJ0211	0.000000	0.000000
19	62011CJ0415	0.039924	0.047619
20	62007CJ0343	0.000000	0.000000
21	62011CJ0092	0.039924	0.142857
22	62002CJ0434	0.000000	0.000000
23	62010CJ0618	0.119772	0.547619
24	62003CJ0210	0.000000	0.000000

... and we visualize for each decision their closeness centrality (blue bars) and betweenness centrality (red bars) scores. From the visualization we can infer that the decision which has the highest 'broker' score also has a high closeness centrality score, and that decisions with a high closeness centrality score do not necessarily have a high betweenness centrality.

```
plt.figure(figsize=(8,8))
sns.barplot(y = df_consprot.cases, x = df_consprot.closeness_norm, color='blue',␣
↪alpha =0.5)
sns.barplot(y = df_consprot.cases, x = df_consprot.betweenness_norm,␣
↪color='red', alpha = 0.5)
```

```
<AxesSubplot:xlabel='betweenness_norm', ylabel='cases'>
```

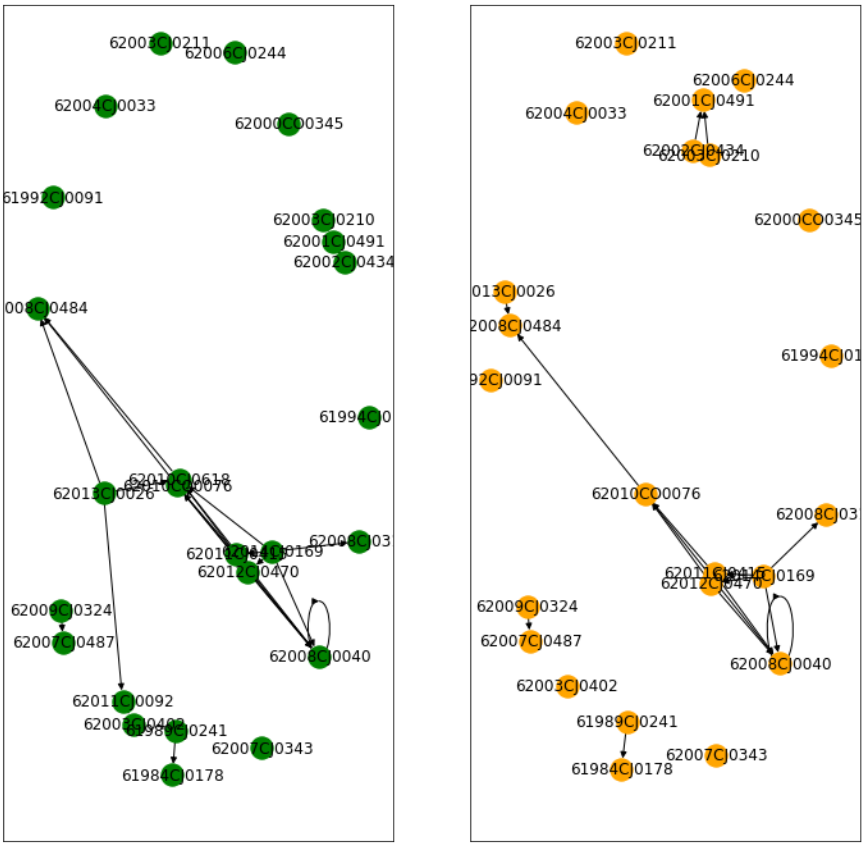


The case with CELEX number C2010CJ0618 is hte decision with the highest betweenness centrality score. Removing these decisions, like in the Medici family example, significantly changes the structure of the network.

```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12,12))

pos = nx.spring_layout(g_consprot2, seed =123)
nx.draw_networkx_nodes(g_consprot2, pos=pos, node_color="green", ax=ax[0])
nx.draw_networkx_edges(g_consprot2, pos=pos, ax=ax[0])
nx.draw_networkx_labels(g_consprot2, pos=pos, ax=ax[0])

g_consprot3 = g_consprot2.subgraph([x for x in g_consprot2.nodes if x not in␣
 ↪["62010CJ0618", "62010CJ0076", "62011CJ0092"]])
pos = nx.spring_layout(g_consprot3, seed =123)
nx.draw_networkx_nodes(g_consprot3, pos=pos, node_color="orange", ax=ax[1])
nx.draw_networkx_edges(g_consprot3, pos=pos, ax=ax[1])
nx.draw_networkx_labels(g_consprot3, pos=pos, ax=ax[1]);
```



How can betweenness centrality be explained in more mathematical terms? Betweenness centrality focuses on the number of shortest paths that pass through a node v. First take any pair of nodes s, t. We can obtain the length of all the shortest paths between s and t using a shortest path function that we here call 'spath'.

$$spath(s, t)$$

We can also get the length of all the shortest paths between s and t that pass through our node of interest, v.

$$spaths(s, t \mid v)$$

(The bar there just says "on condition that they pass through v".)

Subsequently we can calculate the proportion or ratio of shortest paths that pass through v compared to the total number of shortest paths:

$$\frac{spaths(s, t \mid v)}{spath(s, t)}$$

This will do it for a pair of nodes s, t. But we need to do this for every pair of nodes s, t, and sum the results:

$$\sum_{s,t \in V} \frac{spaths(s,t \mid v)}{spath(s,t)}$$

(The Σ is effectively saying, "do this for every pair of nodes s, t in V, adding up the results along the way".)

3.6 Hyperlink-Induced Topic Search (HITS)

Degree, in-degree, and out-degree (centrality) take the direct neighbors into consideration. Closeness and betweenness centrality considers indirect relationships by including the distance of a particular node to other nodes in the network. HITS combines both approaches by considering the number of edges of the nodes that are connected to a particular node. The idea here is that it is important to know whether incoming edge come from or go to nodes that have a lot of edges (or not). Hence, the nodes of a directed graph are measured on two attributes: authority & hub.

- The authority score indicates how frequently a node is cited by nodes that are 'hubs'.
- The hub score, in turn, indicates how often a node is cited by nodes that are 'authorities'.

Both are mutually reinforcing: a good hub is a node that links to many good authorities, and a good authority is a node that is linked to by many good hubs. The authority score is the sum of the hub scores of all nodes that refer to it, whereas the hub score is the sum of the authority scores of all nodes to which it refers.

When analyzing a network of citations between court decision, nodes with a high hub score represent cases that have many citations to authoritative cases, and nodes with a high authority score represent cases that have many citations from hub cases.

The HITS algorithm will return the hub scores and authorities scores for each node. There is also the option to get the results normalized and without normalization.

We apply the algorithm to the obergefell network and we can see what one should expect, that Obergefell is the key hub, and that Loving v. Virginia the key authority.

```
results = nx.hits(g_ob, normalized=False)
results_n = nx.hits(g_ob, normalized=True)
hubs = results[0]
authorities = results[1]
hubs_n = results_n[0]
authorities_n = results_n[1]
df_hits = pd.DataFrame({'nodes':hubs.keys(),
                    'hubs': [np.round(x,2) for x in hubs.values()],
                   'authorities': [np.round(x,2) for x in authorities.values()],
                   'hubs_normalized': [np.round(x,2) for x in hubs_n.values()],
                    "authorities_normalized": [np.round(x,2) for x in␣
 ↪authorities_n.values()]})
df_hits
```

	nodes	hubs	authorities	hubs_normalized	\
0	Obergefell_v_Hodges	-2.77	0.00	0.53	
1	Lawrence_v_Texas	0.00	-0.31	0.00	
2	Bowers_v_Hardwick	0.00	-0.31	0.00	
3	Zablocki_v_Redhail	-0.47	-0.31	0.09	
4	Turner_v_Safley	-0.47	-0.31	0.09	
5	Windsor_v_United_States	-0.47	-0.31	0.09	
6	PPvCasey	-0.48	0.00	0.09	
7	Roe_v_Wade	-0.55	-0.05	0.11	
8	Griswold_v_Connecticut	-0.00	-0.43	-0.00	
9	Lochner_v_New_York	0.00	-0.06	0.00	
10	NAACP_v_State_of_Alabama	0.00	-0.00	0.00	
11	Union_Pacific_Railroad_v_Botsford	0.00	-0.06	0.00	
12	Loving_v_Virginia	0.00	-0.47	0.00	
13	Baker_v_Nelson	0.00	-0.31	0.00	
14	County_of_Sacramento_vs._Lewis	0.00	0.00	0.00	
15	Romer_v_Evans	0.00	0.00	0.00	

	authorities_normalized
0	0.00
1	0.11
2	0.11
3	0.11
4	0.11
5	0.11
6	0.00
7	0.02
8	0.15
9	0.02
10	-0.00
11	0.02
12	0.16
13	0.11
14	0.00
15	0.00

Normalization here means showing each score as the proportion of the total scores, so that each hub score is divided by the sum of hub scores, and each authority score is divided by the sum of authority scores, so for example:

$$hn = \frac{h}{\Sigma_h}$$

Like Eigenvector centrality, HITS is an iterative algorithm that can be represented in many ways. Here, we will present one option that uses repeated multiplication on a modified version of the adjacency matrix. For this, let us first get the adjacency matrix of our graph.

```
Ad = nx.adjacency_matrix(g_ob).todense()
Ad
```

```
matrix([[0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0],
        [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=int32)
```

Now we have to scale the adjacency matrix by the transpose of itself. Transposition here just swaps the order of rows and columns. We call this new matrix M.

$$M = Ad^T \times Ad$$

In numpy code this is represented in this manner:

```
M = Ad.T@Ad
```

Now that we have M, we are going to simulate the repeated scaling of a set of initial values by M, just like with did with Eigenvector Centrality. Since we have a 16x16 matrix, we are going to create a vector x with 16 initial values of 0.0625, which is equivalent to $\frac{1}{16}$.

$$x = [0.0625, 0.0625, 0.0625, 0.0625...]$$

Now we will repeatedly multiply the matrix M for teh vector of probabilities, and normalize the results.

```
x = (np.ones(16)/16)
for i in range(0,100):
    x = M@x.reshape(16,1)
    mynorm = np.linalg.norm(x)
    x = x/mynorm
np.round(x,2)
```

```
array([[0.  ],
       [0.31],
```

```
       [0.31],
       [0.31],
       [0.31],
       [0.31],
       [0.  ],
       [0.05],
       [0.43],
       [0.06],
       [0.  ],
       [0.06],
       [0.47],
       [0.31],
       [0.  ],
       [0.  ]])
```

These results represent our unnormalized authority scores.

Now let's get the hubs scores. These can be gotten by simply "scaling" the authority scores by the original adjacency matrix.

```
y = Ad@x
np.round(y, 2)
```

```
array([[2.77],
       [0.  ],
       [0.  ],
       [0.47],
       [0.47],
       [0.47],
       [0.48],
       [0.55],
       [0.  ],
       [0.  ],
       [0.  ],
       [0.  ],
       [0.  ],
       [0.  ],
       [0.  ],
       [0.  ]])
```

Next we normalize the results by dividing each value (hubs or authorities) by the total sum of values.

```
sumx = np.sum(x)
myauth_norm = x/sumx
myauth_norm = np.round(myauth_norm,2)

sumy = np.sum(y)
```

```
myhubs_norm = y/sumy
myhubs_norm = np.round(myhubs_norm,2)
```

Now that we have the normalized results, we can plug them in our results dataframe and compare. This is just what nx gave us.

```
df_hits['my_hubs_norm'] = myhubs_norm
df_hits['my_auth_norm'] = myauth_norm
df_hits
```

	nodes	hubs	authorities	hubs_normalized	\
0	Obergefell_v_Hodges	-2.77	0.00	0.53	
1	Lawrence_v_Texas	0.00	-0.31	0.00	
2	Bowers_v_Hardwick	0.00	-0.31	0.00	
3	Zablocki_v_Redhail	-0.47	-0.31	0.09	
4	Turner_v_Safley	-0.47	-0.31	0.09	
5	Windsor_v_United_States	-0.47	-0.31	0.09	
6	PPvCasey	-0.48	0.00	0.09	
7	Roe_v_Wade	-0.55	-0.05	0.11	
8	Griswold_v_Connecticut	-0.00	-0.43	-0.00	
9	Lochner_v_New_York	0.00	-0.06	0.00	
10	NAACP_v_State_of_Alabama	0.00	-0.00	0.00	
11	Union_Pacific_Railroad_v_Botsford	0.00	-0.06	0.00	
12	Loving_v_Virginia	0.00	-0.47	0.00	
13	Baker_v_Nelson	0.00	-0.31	0.00	
14	County_of_Sacramento_vs._Lewis	0.00	0.00	0.00	
15	Romer_v_Evans	0.00	0.00	0.00	

	authorities_normalized	my_hubs_norm	my_auth_norm
0	0.00	0.53	0.00
1	0.11	0.00	0.11
2	0.11	0.00	0.11
3	0.11	0.09	0.11
4	0.11	0.09	0.11
5	0.11	0.09	0.11
6	0.00	0.09	0.00
7	0.02	0.11	0.02
8	0.15	0.00	0.15
9	0.02	0.00	0.02
10	-0.00	0.00	0.00
11	0.02	0.00	0.02
12	0.16	0.00	0.16
13	0.11	0.00	0.11
14	0.00	0.00	0.00
15	0.00	0.00	0.00

Remember that repeated multiplication is connected to eigenvectors, so we can infer that an alternative way to get the same result is to get the eigenvectors corresponding to the largest eigenvalue

of M.

```
vals, vecs = np.linalg.eig(M)
```

The largest eigenvalue is in position 2.

```
vals
```

```
array([-2.22044605e-16+0.00000000e+00j,  3.00000000e+00+0.00000000e+00j,
        8.88372254e+00+0.00000000e+00j,  3.44992335e+00+0.00000000e+00j,
        2.31089374e+00+0.00000000e+00j,  1.35546037e+00+0.00000000e+00j,
       -1.07468580e-16+1.71048890e-16j, -1.07468580e-16-1.71048890e-16j,
       -1.82837643e-16+0.00000000e+00j,  1.15584615e-16+0.00000000e+00j,
       -2.38546822e-17+2.77276541e-17j, -2.38546822e-17-2.77276541e-17j,
        2.23686620e-19+0.00000000e+00j,  7.49454261e-34+0.00000000e+00j,
        0.00000000e+00+0.00000000e+00j,  1.00000000e+00+0.00000000e+00j])
```

And this is just our result for the authority score not normalized.

```
a = vecs[:,2]
np.round(a,2)
```

```
matrix([[ 0.  +0.j],
        [ 0.31+0.j],
        [ 0.31+0.j],
        [ 0.31+0.j],
        [ 0.31+0.j],
        [ 0.31+0.j],
        [-0.  +0.j],
        [ 0.05+0.j],
        [ 0.43+0.j],
        [ 0.06+0.j],
        [ 0.  +0.j],
        [ 0.06+0.j],
        [ 0.47+0.j],
        [ 0.31+0.j],
        [-0.  +0.j],
        [-0.  +0.j]])
```

After normalization it is the same as the authority score normalized.

```
sum_a = np.sum(a)
myauth_norm2 = a/sum_a
np.round(myauth_norm2,2)
```

```
matrix([[ 0.  +0.j],
        [ 0.11+0.j],
        [ 0.11+0.j],
```

```
        [ 0.11+0.j],
        [ 0.11+0.j],
        [ 0.11+0.j],
        [-0.  +0.j],
        [ 0.02+0.j],
        [ 0.15+0.j],
        [ 0.02+0.j],
        [ 0.  +0.j],
        [ 0.02+0.j],
        [ 0.16+0.j],
        [ 0.11+0.j],
        [-0.  +0.j],
        [-0.  +0.j]])
```

If we multiply the adjacency matrix for this we can get the hubs score

```
h = Ad@a
h = np.round(h, 2)
h
```

```
matrix([[ 2.77+0.j],
        [-0.  +0.j],
        [ 0.  +0.j],
        [ 0.47+0.j],
        [ 0.47+0.j],
        [ 0.47+0.j],
        [ 0.48+0.j],
        [ 0.55+0.j],
        [ 0.  +0.j],
        [ 0.  +0.j],
        [ 0.  +0.j],
        [ 0.  +0.j],
        [ 0.  +0.j],
        [ 0.  +0.j],
        [ 0.  +0.j],
        [ 0.  +0.j]])
```

. . . and then normalize to end up in the same place through direct application of eigenvectors.

```
sum_h = np.sum(h)
myhubs_norm2 = h/sum_h
np.round(myhubs_norm2,2)
```

```
matrix([[0.53+0.j],
        [0.  +0.j],
        [0.  +0.j],
        [0.09+0.j],
        [0.09+0.j],
```

```
[0.09+0.j],
[0.09+0.j],
[0.11+0.j],
[0.  +0.j],
[0.  +0.j],
[0.  +0.j],
[0.  +0.j],
[0.  +0.j],
[0.  +0.j],
[0.  +0.j],
[0.  +0.j]])
```

The advantages and disadvantages of this or that procedure for getting the results will not be discussed here.

Chapter 4

Community Detection

The detection of communities can bring structure in the network, making the results easier to interpret. In a court decisions citation network, for instance, one rather inspects 35 communities than 2,500 nodes. Ideally, the communities are substantively meaningful, meaning that the court decisions within a community have properties in common that they do not, or to a lesser extent, share with court decisions outside of that community. For instance, court decisions in a specific community in a given network might share that they are about causality, or about traffic accidents, whereas other communities might be about other legal or factual topics. This way, community detection can reveal groups of decisions that concern certain topics, facts, or are otherwise related based on citation patterns.

Community detection methods exist that automatically assign nodes to communities. Below, we give insight into how community detection works.

In order to run the code in this chapter, you will need to import these libraries. The *helper* library can be found in the Github repository for the book.

```
import networkx as nx
import numpy as np
from sklearn.cluster import KMeans
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
from karateclub import Node2Vec
from src.helper import draw_spring, load_graph_from_json
np.random.seed(123)
```

4.1 Machine Learning Methods - KMeans Clustering on Embeddings

One way to find communities is to use machine learning methods on an "embedding" of the network. From a machine learning perspective, the task of finding communities is one of "unsupervised learning", where the algorithm is expected to find patterns in unlabelled data. The network does

not come with ready-made community labels, and unsupervised learning is used to generate them.

In order to use machine learning techniques, however, a network embedding is needed. A network in itself does not have the structure that makes it possible to run a machine learning algorithm on it. Even more, networks may have too many dimensions for an algorithm to run successfully on them. Consequently, one needs to find a mathematical representation of the network that makes it possible to achieve the desired results.

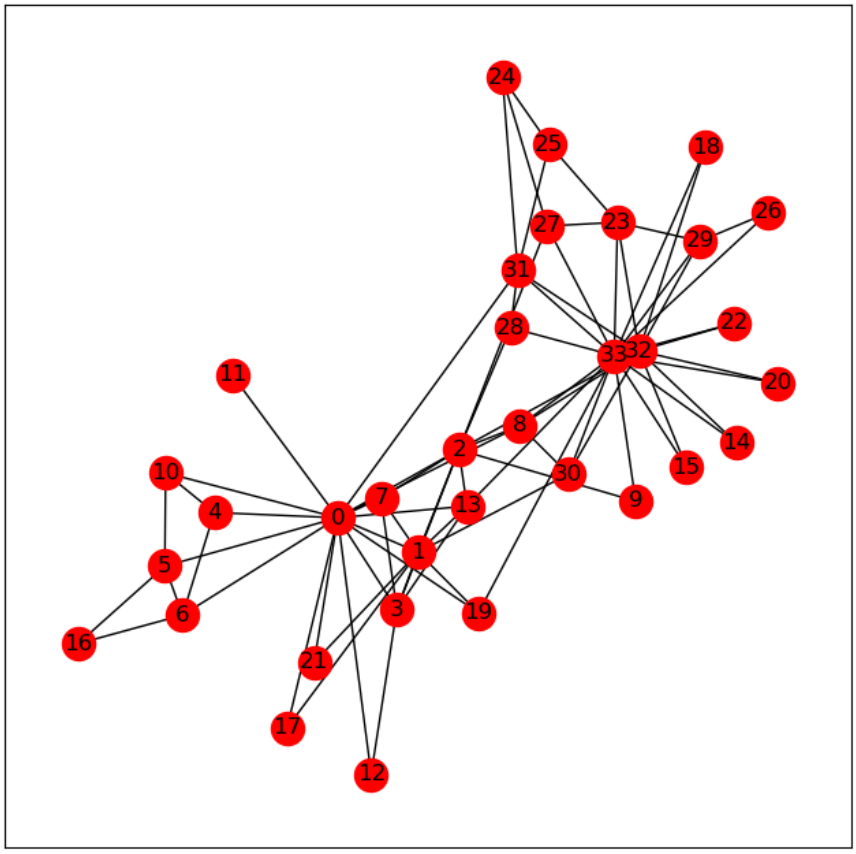
The most common unsupervised learning algorithm is KMeans clustering. In this section, this will be the only algorithm used, and it will be introduced in the next section. However, it should be possible to use virtually any unsupervised learning algorithms on the appropriate network embedding.

Regarding network embeddings, this section will consider the adjacency matrix, the laplacian matrix, the eigenvectors associated with the smallest non-zero eigenvalue and the embedding produced by node2vec. These are all called embeddings here insofar they constitute a mathematical representation of the network. Not all of them are equally good. As we will see, some of these representations lead to unsatisfactory results. But the concept of embeddings is best understood in the progression from the adjacency matrix to more complex representations.

The recurring example will be the classification of the two factions of Zachary's 'karate club' network. The 'karate club' network was recorded by a classic study in network analysis. In 1977 W.W. Zachary published "An Information Model for Conflict and Fission in Small Groups" in the Journal of Anthropological Research. The study is based on observation of a university karate club that would split due to disagreements between the club president and the instructor, leading to the creation of a new club. Network analysis of the friendship relations within the group would serve to predict with more than 97% accuracy who would side with join the group of the President and who would join the group of the instructor (Zachary used Maximum Flow Cut Labelling).

In the networkx version of the dataset, the president is node 0, and the karate instructor node 33. Lets plot the karate club graph here to remember its structure.

```
g_karate = nx.karate_club_graph()
draw_spring(g_karate, node_color="red")
```



4.1.1 KMeans clustering

Kmeans cluster communities are identified by assining nodes to a set of hipothesized clusters. In K-means cluster the K is the hypothesized number of clusters. So if one thinks there are 2 clusters, then k=2, if one things there are 3 clusters, k=3 and so on.

The goal of the K-means clustering technique is to partition space so as to minimize the sum of sqaured residuals from the hipothesized mean within the assumed set of clusters.

SSR is defined as:

$$SSR = \sum(x - \mu)$$

Kmeans records the SSR for every cluster, and tries to mimize the value of SSR over all the hypothesized k clusters

$$argmin(\sum_{i}^{k}\sum(x - \mu))$$

Note the precedence of the summartion operators Σ. The first summation operator does the SSR within each cluster. The second summation operator sums the results for each of the clusters.

(We leave out of consideration whether the SSR should be normalized or not)

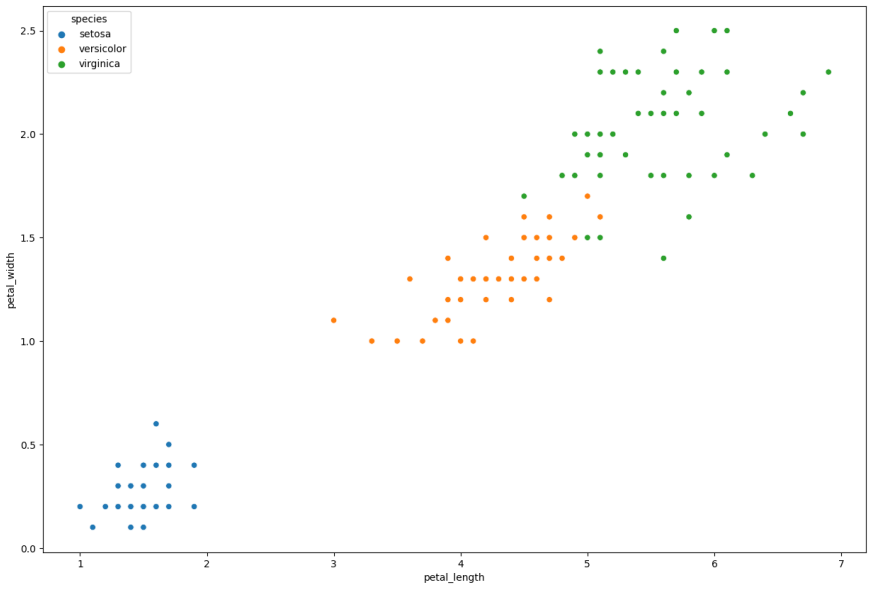
We speak about hypothesized clusters and means because the way that Kmeans works (in its most basic form) is by:

1) First hypothesizing a mean for each of the cluster randomly.
2) Assigning each data point to the mean that is closest to it
3) Measuring the SSR score
4) Calculating new means on the basis of the existing clusters.
5) Repeat from point 2...
6) Until data points no longer change clusters (convergence), a specified number of iteration or meeting a tolerance threshold

Below we can see KMeans in action in a traditional dataset. The iris dataset records the petal lenght, petal width, sepal length and sepal width of 3 species of iris flowers. It can be seen that -for example- the size and with of the petals is highly correlated and is highly informative of what species of flower we are looking at.

```
iris = sns.load_dataset("iris")
iris.head()
sns.scatterplot(data=iris, x="petal_length", y="petal_width", hue="species")
```

```
<AxesSubplot: xlabel='petal_length', ylabel='petal_width'>
```



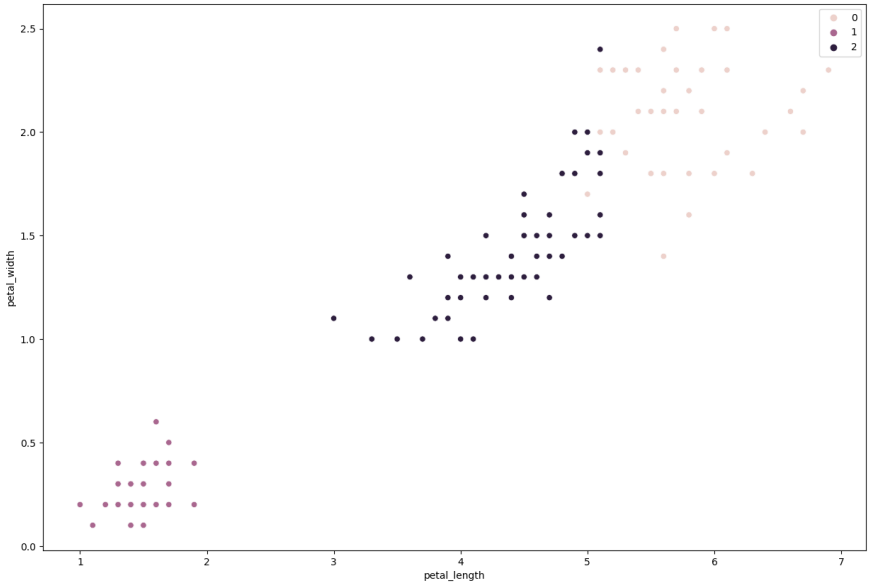
But lets imagine that we do not have the labels to begin with. KMeans can be used to divide the dataset into 3 parts. As we can see from the plot below, it reaches an almost perfect match to the actual taxonomic classification.

Here we will run the algorithm in verbose form to see how SSR decreases with each step.

```
km = KMeans(n_clusters=3, verbose=True, n_init=1).fit(iris.drop("species",␣
↪axis=1))
predictions = km.predict(iris.drop("species", axis=1))
sns.scatterplot(data=iris, x="petal_length", y="petal_width", hue=predictions)
```

```
Initialization complete
Iteration 0, inertia 144.03
Iteration 1, inertia 79.3943288531747
Iteration 2, inertia 78.910056736457
Iteration 3, inertia 78.85144142614601
Converged at iteration 3: strict convergence.
```

```
<AxesSubplot: xlabel='petal_length', ylabel='petal_width'>
```



4.1.2 Clustering on the adjacency matrix and the laplacian matrix

Now lets try to get the two groups using KMeans clustering on the adjacency matrix and the laplacian matrix.

This is a good time to recall a variety of matrices that help represent the information on a graph. First we have the degree matrix. This records on every degree of every node in the diagonal.

$$D = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

The adjacency matrix, by contrast, records the edges between the nodes. So a 1 here means a link from a node to another and a 0 none (this was explained in more detail in section 3.3).

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

Using these two matrices we can find the Graph Laplacian matrix by simply subtracting the adjacency matrix from the degree matrix, so

$$L = D - A$$

Substraction of matrices is basic arithmetic, it just means elementwise substractions, so we do (for the first row:) 2-0, 0-1, 0-1, 0-0, and so forth until we get:

$$L = \begin{pmatrix} 2 & -1 & -1 & 0 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 3 & -1 \\ 0 & -1 & -1 & 2 \end{pmatrix}$$

Naturally the arithmetic involved in this will be practically impossible for humans on any matrix of reasonable size. Software can just give you the Laplacian matrix of a graph on command.

Now lets try to run KMeans clustering on these matrices that we are going to regard as the most naive embeddings of the network.

Using either the adjacency matrix or the graph matrix does not succeed. At least the president (node 0) and the karate instructor (33) are put in different groups, but the students are not correctly classified in their respective clicques.

Arguably the reason this does not work is that the adjacency matrix and the laplacian matrix are poor representations of the graph for machine learning. While the network can be reconstructed from these matrices, from a machine learning perspective all that these representations record is the neighbors that are one jump away from each other, and degree information if it is a laplacian matrix.

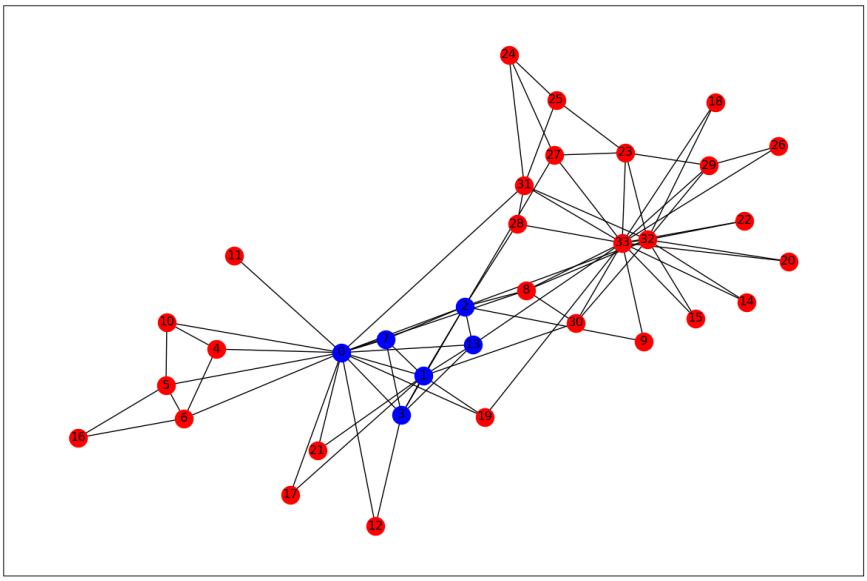
```
g_karate = nx.karate_club_graph()
M = nx.adjacency_matrix(g_karate).todense()

km = KMeans(2)
res = km.fit_predict(M)

reszero = np.where(res==0)[0].tolist()
```

```
resone = np.where(res==1)[0].tolist()

pos = nx.spring_layout(g_karate, seed=123)
nx.draw_networkx_nodes(g_karate, pos = pos, nodelist=reszero, node_color="blue")
nx.draw_networkx_nodes(g_karate, pos = pos, nodelist=resone, node_color="red")
nx.draw_networkx_edges(g_karate, pos = pos)
nx.draw_networkx_labels(g_karate, pos = pos);
```



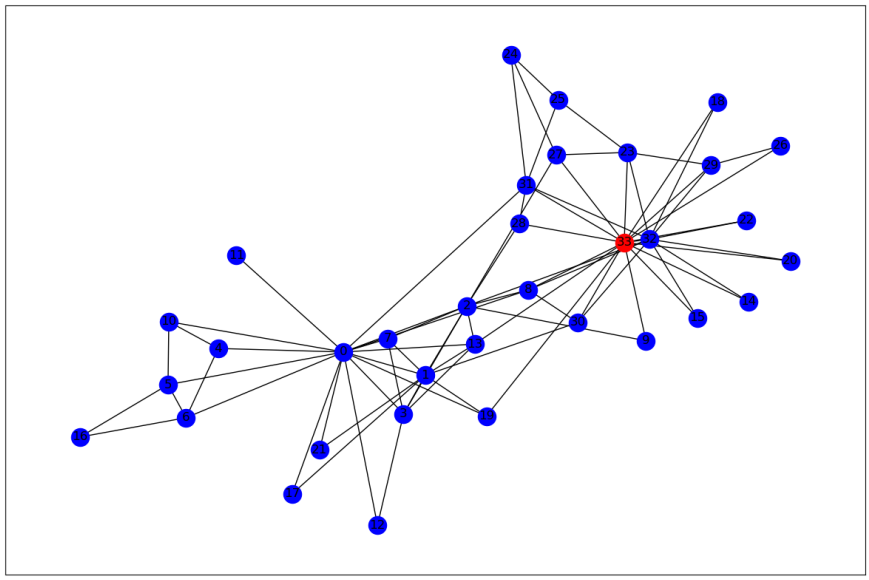
The results are worse if we use the laplacian matrix.

```
M = nx.laplacian_matrix(g_karate).todense()


km = KMeans(2)
res = km.fit_predict(M)

reszero = np.where(res==0)[0].tolist()
resone = np.where(res==1)[0].tolist()

pos = nx.spring_layout(g_karate, seed=123)
nx.draw_networkx_nodes(g_karate, pos = pos, nodelist=reszero, node_color="blue")
nx.draw_networkx_nodes(g_karate, pos = pos, nodelist=resone, node_color="red")
nx.draw_networkx_edges(g_karate, pos = pos)
nx.draw_networkx_labels(g_karate, pos = pos);
```



4.1.3 Spectral clustering

The problem with the above attempt is that the adjacency matrix or laplacian is a not a good network embedding for the purposes of clustering. It fails to represent "what is going on" inside the network.

A better approach would be spectral clustering. Here isntead of using either one of the aforementioned matrixes, we apply clustering to certain mathematic properties that arise from the laplacian matrix, through so-called spectral decomposition.

First we need the laplacian matrix of the karate club graph.

```
laplacian_karate = nx.laplacian_matrix(g_karate)
```

The next step is to find the eigenvector associated with the n smallest non-zero eigenvalues. The n depends on how many clusters you want. If you want two clusters, you need to look for the smallest eigenvector. If you want three clusters, you select the two smallest, and so on.

Let us print a list of eigenvalues (obtained with Numpy). Note that there are 34 eigenvalues, one for each node in the Karate club network example.

```
# this get us the values and the vectors
vals, vecs = np.linalg.eig(laplacian_karate.todense())
# here we extract the values
vals
```

```
array([ 5.20653410e+01,  4.59907681e+01,  4.18328632e+01,  3.79927558e+01,
        2.96883840e+01,  2.55555039e+01,  2.27849630e+01,  2.13586504e+01,
```

```
       1.93238240e+01,  1.70337951e+01,  1.48885752e+01,  1.43813508e+01,
       1.20754027e+01,  1.18744202e+01, -2.37228377e-15,  1.02218147e+01,
       1.00106981e+01,  9.18096651e+00,  1.18710730e+00,  6.97788032e+00,
       6.44605541e+00,  5.59481244e+00,  2.39431926e+00,  5.16855490e+00,
       4.97817925e+00,  4.69727550e+00,  4.54469763e+00,  4.16160574e+00,
       3.70606484e+00,  3.80094429e+00,  2.93182048e+00,  2.96830180e+00,
       3.12126310e+00,  3.06104079e+00])
```

Lets sort the values to make it easier to spot the second smallest eigenvalue.

```
np.sort(vals)
```

```
array([-2.37228377e-15,  1.18710730e+00,  2.39431926e+00,  2.93182048e+00,
       2.96830180e+00,  3.06104079e+00,  3.12126310e+00,  3.70606484e+00,
       3.80094429e+00,  4.16160574e+00,  4.54469763e+00,  4.69727550e+00,
       4.97817925e+00,  5.16855490e+00,  5.59481244e+00,  6.44605541e+00,
       6.97788032e+00,  9.18096651e+00,  1.00106981e+01,  1.02218147e+01,
       1.18744202e+01,  1.20754027e+01,  1.43813508e+01,  1.48885752e+01,
       1.70337951e+01,  1.93238240e+01,  2.13586504e+01,  2.27849630e+01,
       2.55555039e+01,  2.96883840e+01,  3.79927558e+01,  4.18328632e+01,
       4.59907681e+01,  5.20653410e+01])
```

Note that numpy will often give you very small numbers which are equivalent to zero. Here the first number is practically a zero. We can see this if we round the numbers to, say, 5 decimal places.

```
np.sort(np.round(vals,5))
```

```
array([-0.     ,  1.18711,  2.39432,  2.93182,  2.9683 ,  3.06104,
       3.12126,  3.70606,  3.80094,  4.16161,  4.5447 ,  4.69728,
       4.97818,  5.16855,  5.59481,  6.44606,  6.97788,  9.18097,
      10.0107 , 10.22181, 11.87442, 12.0754 , 14.38135, 14.88858,
      17.0338 , 19.32382, 21.35865, 22.78496, 25.5555 , 29.68838,
      37.99276, 41.83286, 45.99077, 52.06534])
```

Now we can see that the smallest non-zero eigenvalue is 1.18710730e+00. To avoid human errors with counting, we can use `np.where` to find its position.

It tells us that it is the position 18 in the array.

```
# because python indexes start at 0, indexing at 1 will give us the second␣
 ↪smallest value.
smallest_non_zero = np.sort(vals,)[1]
np.where(vals==smallest_non_zero)
```

```
(array([18], dtype=int64),)
```

Which we can confirm as true

```
vals[18]
```

```
1.1871073019962068
```

Now we need to get the eigenvector associated with the smallest non-zero eigenvalue, which is the eigenvector in position 18. Here we just need to index into the 18th column (counting from zero) of the eigenvector variable.

```
X = vecs[:,18]
X = np.asarray(X)
X
```

```
array([[ 0.1233171 ],
       [ 0.0580066 ],
       [ 0.01368446],
       [ 0.07445083],
       [ 0.26733837],
       [ 0.29881441],
       [ 0.29744558],
       [ 0.06406144],
       [-0.0530052 ],
       [-0.12925543],
       [ 0.28550172],
       [ 0.20406685],
       [ 0.12324309],
       [ 0.03493189],
       [-0.17156602],
       [-0.15520216],
       [ 0.37166422],
       [ 0.16804128],
       [-0.21148231],
       [ 0.0625882 ],
       [-0.18847412],
       [ 0.1289233 ],
       [-0.16858175],
       [-0.14801794],
       [-0.15762212],
       [-0.14632034],
       [-0.19249255],
       [-0.12982346],
       [-0.09449019],
       [-0.16960883],
       [-0.08368282],
       [-0.1170647 ],
       [-0.13538405],
       [-0.12400534]])
```

Having this, we can cluster them using KMeans. In our case, we select two clusters because we know from the paper by Zachary that the group would split into two.

In order to run KMeans first we need to turn the array X into a two dimensional array as sklearn

expects this type of data. (It may seem surprising that a single column should be considered two dimensional. A one dimensional array does not specify if the array is a row or a column. A two dimensional array will specify if it is a row or a column, even if it has only one row or only one column. In this sense it is "two dimensional").

```
# the -1 here says "give me as many rows as necessary", the 1 says "and just one␣
↪column"
X = X.reshape(-1,1)
print(X.shape)
```

```
(34, 1)
```

Before running KMeans, if we inspect the vector visually we may see that it is possible to see two clusters, one of positive signed numbers and one of positive signed numbers.

```
np.sort(X, axis=0)
```

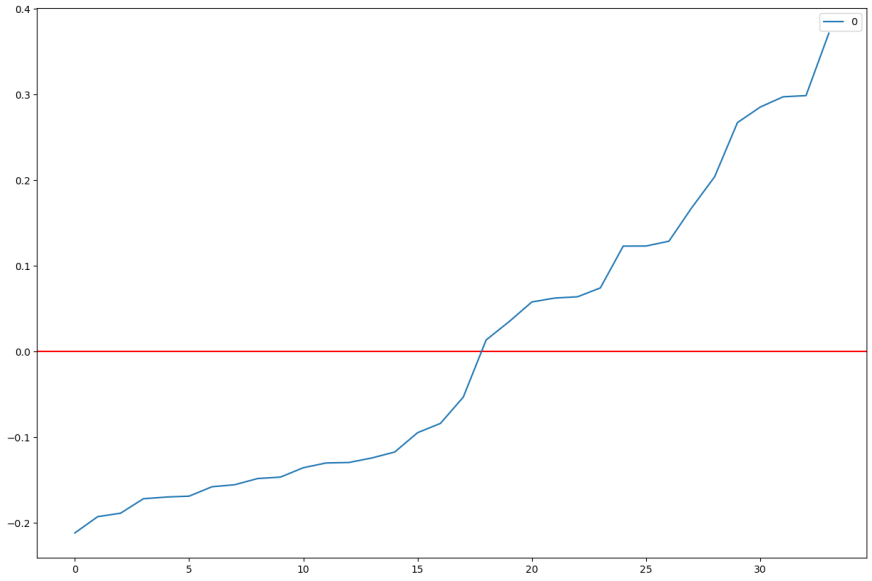
```
array([[-0.21148231],
       [-0.19249255],
       [-0.18847412],
       [-0.17156602],
       [-0.16960883],
       [-0.16858175],
       [-0.15762212],
       [-0.15520216],
       [-0.14801794],
       [-0.14632034],
       [-0.13538405],
       [-0.12982346],
       [-0.12925543],
       [-0.12400534],
       [-0.1170647 ],
       [-0.09449019],
       [-0.08368282],
       [-0.0530052 ],
       [ 0.01368446],
       [ 0.03493189],
       [ 0.0580066 ],
       [ 0.0625882 ],
       [ 0.06406144],
       [ 0.07445083],
       [ 0.12324309],
       [ 0.1233171 ],
       [ 0.1289233 ],
       [ 0.16804128],
       [ 0.20406685],
       [ 0.26733837],
       [ 0.28550172],
```

```
       [ 0.29744558],
       [ 0.29881441],
       [ 0.37166422]])
```

Or more visually...

```
plt.axhline(0, color="red")
sns.lineplot(np.sort(X, axis=0))
```

```
<AxesSubplot: >
```



Now we run KMeans with the requirement of finding two clusters

```
from sklearn.cluster import KMeans
km = KMeans(2)

# Kmeans will return an array, but networkx (in the next cell) will want a list.
# Therefore we need to conver to a list, and get the indexes (the positions) in␣
 ↪the list where our results lie

results = list(km.fit_predict(X))


nodes_cluster_1 = [x[0] for x in enumerate(results) if x[1]==0]
print("nodes for cluster 1", nodes_cluster_1,"\n")
nodes_cluster_2 = [x[0] for x in enumerate(results) if x[1]==1]
print("nodes for cluster 2", nodes_cluster_2,"\n")
```

```
nodes for cluster 1 [8, 9, 14, 15, 18, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30,
```

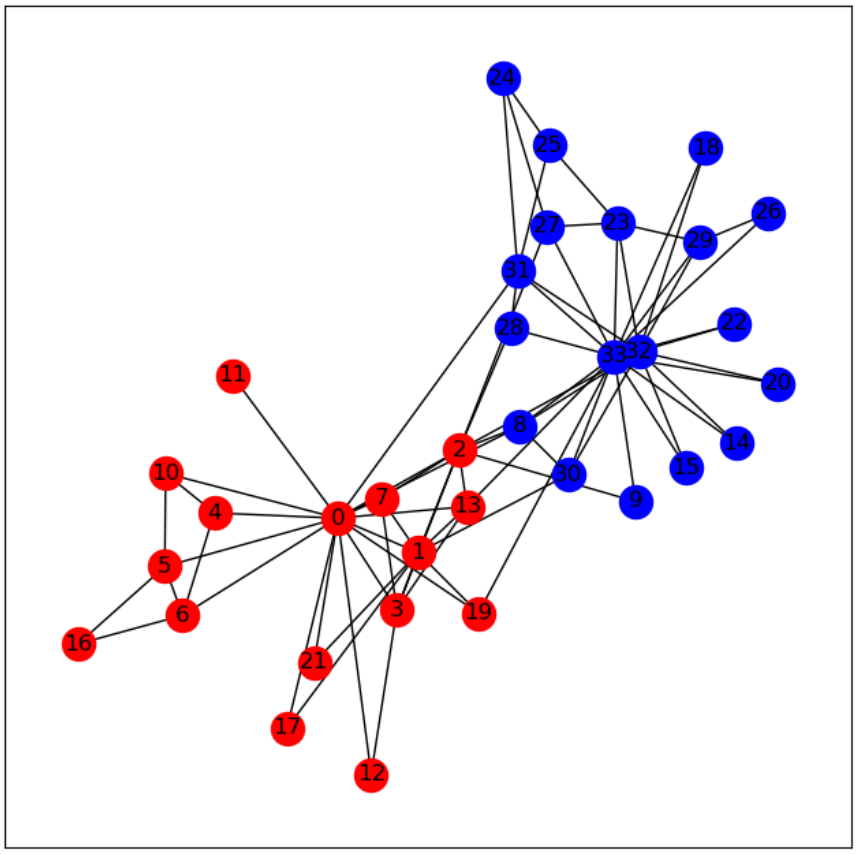
```
31, 32, 33]

nodes for cluster 2 [0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 16, 17, 19, 21]
```

And here we can see the results of the clustering, which give you a very reasonable partition of the network.

```
plt.figure(figsize=(8,8))
pos = nx.spring_layout(g_karate, seed =123)
nx.draw_networkx_nodes(g_karate, pos=pos, nodelist=list(nodes_cluster_1),␣
↪node_color="blue")
nx.draw_networkx_nodes(g_karate, pos=pos, nodelist=list(nodes_cluster_2),␣
↪node_color="red")
nx.draw_networkx_labels(g_karate, pos=pos)
nx.draw_networkx_edges(g_karate, pos=pos)
```

```
<matplotlib.collections.LineCollection at 0x1a647971df0>
```



4.1.4 Node2Vec

A final, more modern approach to consider now is Node2Vec. This is an algorithm that produces graph embeddings. That is, it turns a graph into a mathematical representation that has some high degree of fidelity to what is going on in the network. Explaining Node2Vec is beyond the scope of this work. However we can instantiate it and see it in action. Here we will represent the karate club graph in 3 dimensions (the default is 128 dimensions, so this is a highly watered down version). In practice, node2vec will simplify the graph by giving each of the 34 nodes 3 values. This can be seen in the array below:

```
n2v = Node2Vec(dimensions=3)
n2v.fit(g_karate)
X = n2v.get_embedding()
X
```

```
array([[-0.39774638,  0.98552513,  1.549921  ],
       [-0.08955019,  1.1593839 ,  1.1336117 ],
       [ 0.53977144,  1.3380251 ,  0.41468266],
       [-0.37382326,  1.0306491 ,  1.1607355 ],
       [-0.89236057,  1.3442596 ,  2.0145543 ],
       [-0.7951624 ,  1.5147979 ,  3.1427975 ],
       [-1.1356802 ,  1.2860818 ,  2.4611368 ],
       [-0.0718383 ,  0.74390393,  1.211727  ],
       [ 0.7060882 ,  1.1848918 , -0.5075911 ],
       [ 0.47521508,  1.3774163 , -0.31074014],
       [-1.1328117 ,  1.454686  ,  2.9339418 ],
       [-2.098106  ,  2.354874  ,  1.5865073 ],
       [-0.06156549,  0.9463011 ,  1.369508  ],
       [-0.25764835,  1.1727036 ,  0.8396442 ],
       [ 0.65459263,  1.6293339 , -1.09808   ],
       [ 0.64556366,  1.678741  , -1.9491938 ],
       [-1.3291464 ,  1.4751254 ,  3.2024274 ],
       [-0.37588575,  1.6447046 ,  1.0192587 ],
       [ 0.84376836,  1.828418  , -1.514843  ],
       [ 0.09403782,  1.1426826 ,  0.9819763 ],
       [ 1.1992661 ,  1.8492862 , -2.0431974 ],
       [-0.2687595 ,  1.3410224 ,  1.2516799 ],
       [ 1.0863756 ,  1.8846724 , -1.6142293 ],
       [ 2.2534754 ,  0.83907664, -0.5418599 ],
       [ 3.3801103 ,  0.46167666,  0.33384326],
       [ 3.7947485 ,  0.5942839 ,  0.64808065],
       [ 1.1298275 ,  1.9480387 , -1.5947171 ],
       [ 2.416353  ,  0.69496584, -0.03045547],
       [ 0.8184565 ,  0.95051855, -0.3160597 ],
       [ 1.7769817 ,  1.4512491 , -0.85625404],
       [ 0.9644101 ,  1.3025742 , -0.5567254 ],
       [ 2.2199378 ,  0.64713514,  0.12074987],
```

```
       [ 0.8884901 ,  2.0890234 , -1.6045245 ],
       [ 1.3918453 ,  1.9844021 , -1.2040191 ]], dtype=float32)
```

Now we run the Kmeans algorithm on the embedding.

```
km = KMeans(n_clusters=2)
results = km.fit_predict(X)
nodes_cluster_1 = [x[0] for x in enumerate(results) if x[1]==0]
print("nodes for cluster 1", nodes_cluster_1,"\n")
nodes_cluster_2 = [x[0] for x in enumerate(results) if x[1]==1]
print("nodes for cluster 2", nodes_cluster_2,"\n")
```

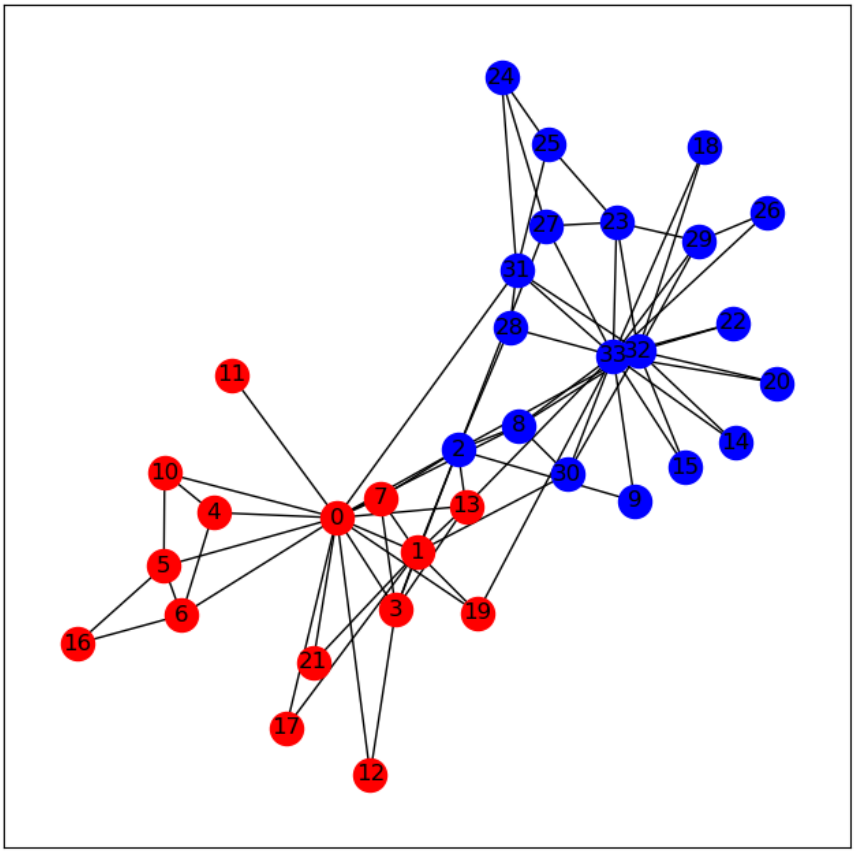
```
nodes for cluster 1 [2, 8, 9, 14, 15, 18, 20, 22, 23, 24, 25, 26, 27, 28, 29,
30, 31, 32, 33]

nodes for cluster 2 [0, 1, 3, 4, 5, 6, 7, 10, 11, 12, 13, 16, 17, 19, 21]
```

And the result is a very reasonable approximation to the split between the two groups.

```
g_karate = nx.karate_club_graph()
plt.figure(figsize=(8,8))
pos = nx.spring_layout(g_karate, seed =123)
nx.draw_networkx_nodes(g_karate, pos=pos, nodelist=nodes_cluster_1,␣
 ↪node_color="blue")
nx.draw_networkx_nodes(g_karate, pos=pos, nodelist=nodes_cluster_2,␣
 ↪node_color="red")
nx.draw_networkx_labels(g_karate, pos=pos)
nx.draw_networkx_edges(g_karate, pos=pos)
```

```
<matplotlib.collections.LineCollection at 0x1a64d7b5eb0>
```



4.2 Network Algorithms - Louvain Communities

In opposition to machine learning methods we can find network algorithms. These do not transform the graph into an embedding, to then run traditional machine learning algorithms (like KMeans clustering). Rather they work with the existing graph and its properties to find clusters.

In relation to network algorithms we will consider only the Louvain method for community detection. This is an algorithm that finds communities by using the concept of modularity gain.

In order to fully introduce this method, we first have to understand the concept of modularity.

4.2.1 Modularity

Modularity in essence does two things: 1. It compares the edges of nodes in a certain community to the edges the nodes have to nodes outside of the community. Court decisions on contract law are more likely to cite one another (edges within the community) than to cite court decisions on human rights (edges outside of the community). The edges within the community can be compared to the edges outside of the community. The higher the number of edges within the community relative

to the edges with nodes outside of the community, the higher the modularity. 2. It takes into consideration the number of edges in the network. The higher the number of edges in the network, the less important the difference becomes between the edges within the community and the edges to nodes outside of the community. This prevents small yet dense communities obtaining the highest modularity score and makes it more likely that larger denser communities (with more edges) will receive a high modularity score.

Both considerations are reflected in the formula for modularity, For ease of presentation, we will only consider modularity for undirected networks.

The formula for modularity can be presented in a variety of ways and there are advantages to each form or presentation. Here are two formulas:

$$Q = \frac{1}{2m} \sum_{c=1}^{n} \left[\Sigma A_c - \gamma \frac{\Sigma D_c^2}{2m} \right]$$

$$Q = \sum_{c=1}^{n} \left[\frac{\Sigma A_c}{2m} - \gamma \left(\frac{\Sigma D_c}{2m} \right)^2 \right]$$

(These formulas have been adapted from the documentation to make the notation easier to follow, especially for an attempt to put them in practice by hand.)

For our discussion we will focus on the second formula, because it most clearly shows a difference between two ratios. The representation we choose here provides the most insight into what modularity does.

- ΣA_c is the sum of the adjacency matrix of the community only (no edges to nodes outside of the community) .
- m is the total number of edges of the network.
- $SigmaD_c$ is the degree of the nodes of the community, taking into consideration their edges to nodes outside of the community.
- γ is a dampening parameter, that for our purposes can be ignored if set to 1. Making this number smaller will make it easier to find communities, and making it larger makes communities harder to find.

We observe a difference between fractions. That is to say, a difference between proportions.

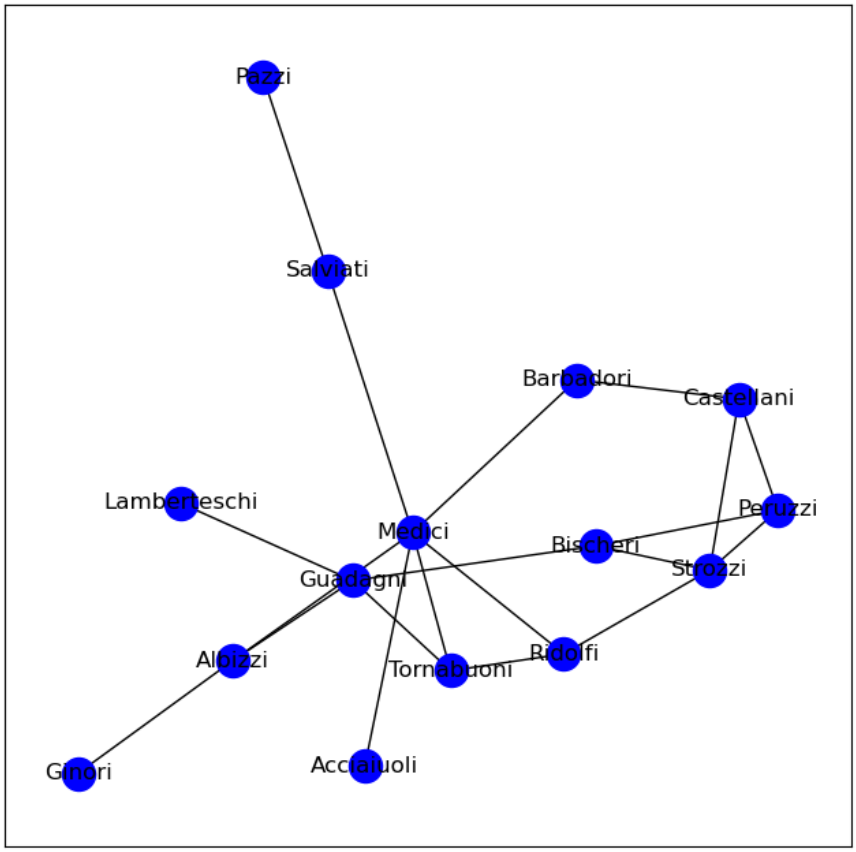
- $\frac{\Sigma A_c}{2m}$ represents the proportion of purely intra-community edges for a given community to the total number of edges. The division by 2 considers the fact that the adjacency matrix will count two for every edge.
- $\left(\frac{\Sigma D_c}{2m}\right)^2$ represents the proportion of the edges that a community has to nodes within and outside of itself to the total number of edges in the network.

The logic of this is that if the left hand side of the difference is greater than the right hand side of the difference one will get a positive number. And the larger the difference, the tighter a community is, the larger the number.

- The Σ outside the outer brackets is a summation notation that states that the difference between fractions will be repeated for every hypothesized community and added up. The higher the ultimate summation, the better the community partition is.

We can operationalize the formula "by hand" on the graph of Florentine families hypothesizing that the families Castellani, Peruzzi and Strozzi form a community vis-a-vis the rest of the other families. A quick look at the graphs suggest that this is a reasonable grouping.

```
g_florentine = nx.florentine_families_graph()
draw_spring(g_florentine)
```



First let us save two lists with the names of the nodes that we want to use as hypothesized communities. Castellani, Peruzzi and Strozzi in one list, the rest on the other.

```
fl_CPS = ['Castellani','Peruzzi','Strozzi']
fl_other= ['Acciaiuoli', 'Medici', 'Barbadori', 'Ridolfi', 'Tornabuoni',␣
↪'Albizzi', 'Salviati', 'Pazzi', 'Bischeri', 'Guadagni', 'Ginori',␣
↪'Lamberteschi']
```

Since m is just the number of edges we can obtain the number in this manner.

```
m = len(g_florentine.edges)
m
```

20

For ΣA_c we need to get the adjacency matrix of just the nodes within the community. This means we need to sever the connection of the community with other nodes. We can get this with the subgraph function.

```
sub_fl_CPS = g_florentine.subgraph(fl_CPS)
sub_fl_other = g_florentine.subgraph(fl_other)
```

Now we can get the adjacency matrix of the subgraphs like this...

```
nx.adjacency_matrix(sub_fl_CPS).todense()
```

```
matrix([[0, 1, 1],
        [1, 0, 1],
        [1, 1, 0]], dtype=int32)
```

Numpy can sum all the values of such matrices with a call to np.sum.

```
SA_cps = np.sum(nx.adjacency_matrix(sub_fl_CPS).todense())
SA_other = np.sum(nx.adjacency_matrix(sub_fl_other).todense())
print(f"the value of SigmaA_c for the CPS community is {SA_cps} and for the␣
 ↪other is {SA_other}")
```

```
the value of SigmaA_c for the CPS community is 6 and for the other is 26
```

This gets us all we need for the left hand side of the difference.

Now we move to work on the right side of the difference. The value of $ ΣD_c $ is the sum of the degree of the nodes that interest us, but not with their values cut off from the network -that is, not as subgraphs- but in their full values connected to the Florentine families network.

In terms of code it can be slightly tricky to unpack these values. Below we use a np.sum over a list comprehension that produces the degree scores of only the nodes we need.

```
SD_cps = np.sum([x[1] for x in list(nx.degree(g_florentine, fl_CPS))])

SD_other = np.sum([x[1] for x in list(nx.degree(g_florentine, fl_other))])

print(f"the value of SigmaD_c for the CPS community is {SD_cps} and for the␣
 ↪other is {SD_other}")
```

```
the value of SigmaD_c for the CPS community is 10 and for the other is 30
```

Now we are ready to fill in the blanks. Remember the formula is:

$$Q = \sum_{c=1}^{n} \left[\frac{\Sigma A_c}{2m} - \gamma \left(\frac{\Sigma D_c}{2m} \right)^2 \right]$$

So for the first community we have

$$CPS = \left[\frac{6}{40} - \gamma \left(\frac{10}{40} \right)^2 \right] = 0.875$$

```
CPS_result = (SA_cps/(2*m)) - ((SD_cps/(2*m))**2)
CPS_result
```

```
0.0875
```

And for the second

$$other = \left[\frac{26}{40} - \gamma\left(\frac{30}{40}\right)^2\right] = 0.875$$

```
other_result = (SA_other/(2*m)) - ((SD_other/(2*m))**2)
other_result
```

```
0.08750000000000002
```

And adding them together we get 0.175

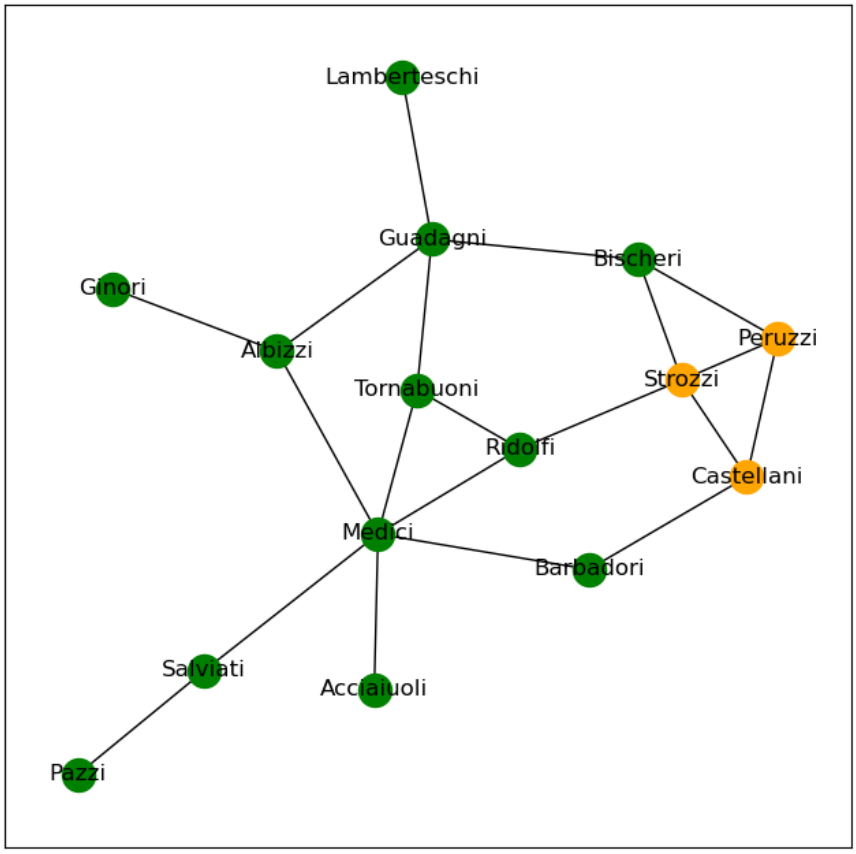
```
CPS_result + other_result
```

```
0.17500000000000002
```

Which is the same as the official result.

Thus the following partition has the positive modularity score of 0.175.

```
plt.figure(figsize=(8,8))
pos = nx.spring_layout(g_florentine)
nx.draw_networkx_nodes(g_florentine, pos, nodelist=fl_CPS, node_color='orange')
nx.draw_networkx_nodes(g_florentine, pos, nodelist=fl_other, node_color='green')
nx.draw_networkx_edges(g_florentine, pos=pos)
nx.draw_networkx_labels(g_florentine, pos=pos);
```



4.2.2 Louvain Communities

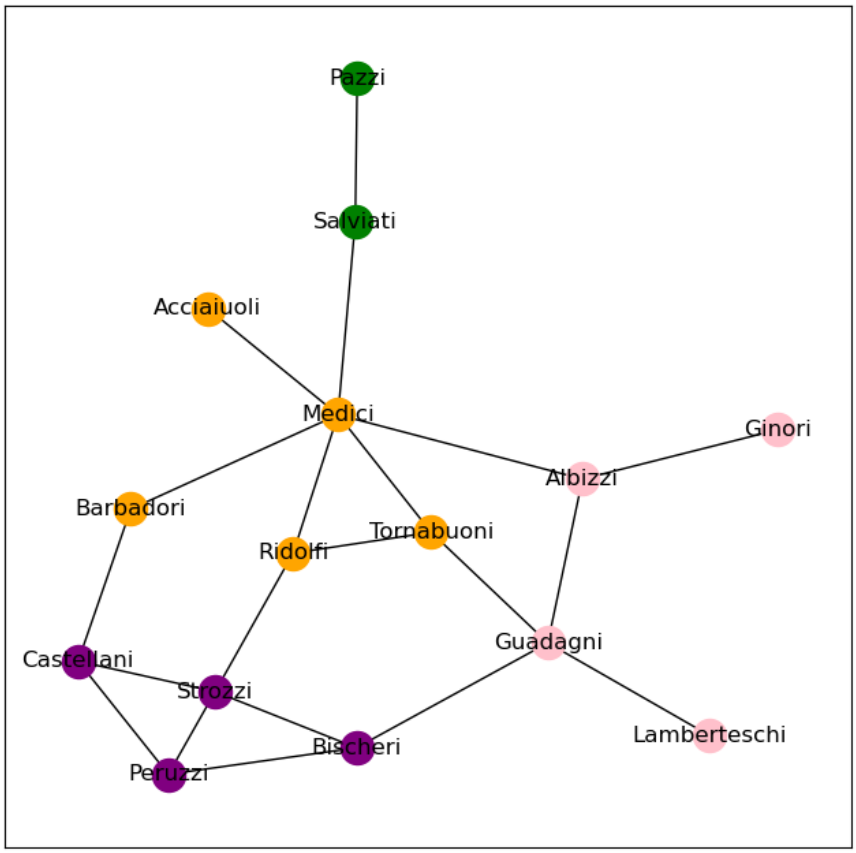
What Louvain community detection does, is to try to find the distribution of communities that will give the highest modularity score, using a sort of cumulative random search.

1. It will asume every node is a community and score the modularity.
2. It will assign every node to a community with one of its neighbors and score the modularity of the result.
3. It will keep the assignment that gives the highest modularity score.
4. The resulting communities will be treated as nodes.
5. Step 1 is repeated.

This process keeps going until there is no more modularity gain.

We can quickly try to run the Louvain communities algorithm to find the best partition of the Florentine families network that we used earlier to discuss modularity. It turns out the best split is in four communities. Compared to our intuitive results presented above, we can see that Castellani, Peruzzi and Strozzi did deserve to go together, but that Bischeri should have been also part their team.

```
plt.figure(figsize=(8,8))
result = nx.algorithms.community.louvain_communities(g_florentine)
pos = nx.spring_layout(g_florentine)
nx.draw_networkx_nodes(g_florentine, pos, nodelist=result[0],␣
↪node_color='orange')
nx.draw_networkx_nodes(g_florentine, pos, nodelist=result[1], node_color='green')
nx.draw_networkx_nodes(g_florentine, pos, nodelist=result[2],␣
↪node_color='purple')
nx.draw_networkx_nodes(g_florentine, pos, nodelist=result[3], node_color='pink')
nx.draw_networkx_edges(g_florentine, pos=pos)
nx.draw_networkx_labels(g_florentine, pos=pos);
```



Now let us calculate the modularity score for the Louvain-detected communities. We can see that at 0.3975 it is quite a bit higher than our intuitive partition.

```
nx.algorithms.community.modularity(g_florentine, result)
```

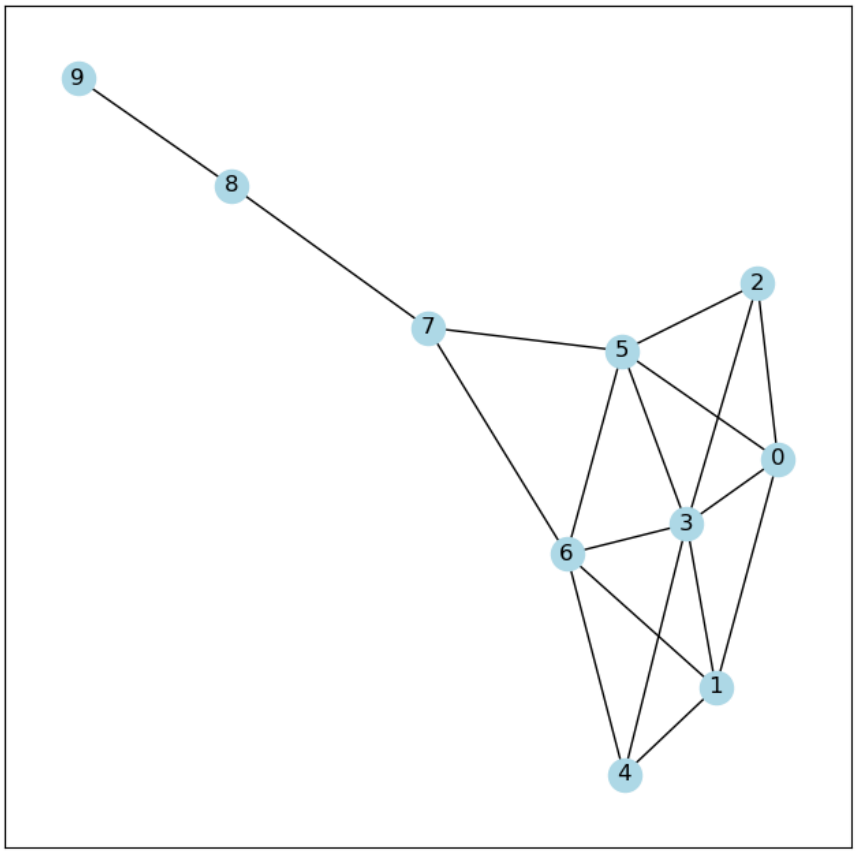
```
0.39749999999999996
```

We now try to approximate this by hand. Let us see how far we can get. We will use a smaller graph, the kite, to make our life easier. There are key differences between what we are doing and

the actual Louvain Communities algorithm. However, this will be enough for us to get an idea of what is going on.

The question is how to best partition the kite in terms of modularity.

```
g_kite = nx.krackhardt_kite_graph()
draw_spring(g_kite, node_color="lightblue")
```



Doing a pseudo-Louvain process, We begin by assigning each node in the kite graph to its own community. Since this is a very bad partition, it makes sense that we get a negative modularity score.

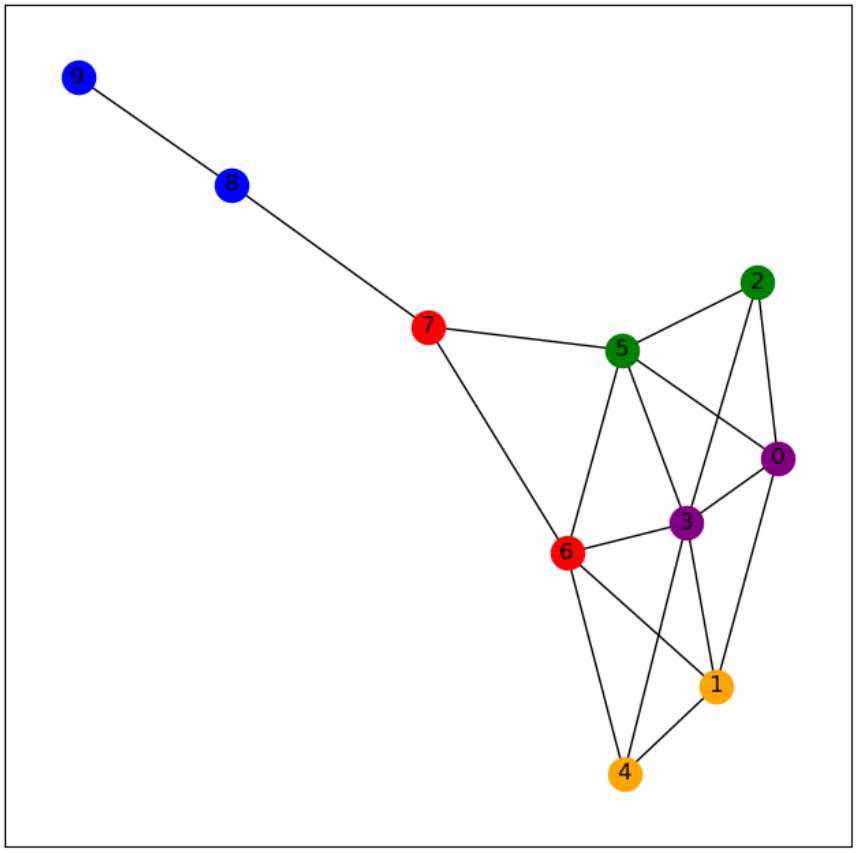
```
nx.algorithms.community.modularity(g_kite, [[x] for x in range(0,10)])
```

```
-0.11574074074074076
```

Now we will assign each node, to one of its neighbors, and obtain the modularity score.

```
plt.figure(figsize=(8,8))
pos = nx.spring_layout(g_kite, seed=123)
nx.draw_networkx_nodes(g_kite, pos, nodelist=[9,8], node_color='blue')
```

```
nx.draw_networkx_nodes(g_kite, pos, nodelist=[7,6], node_color='red')
nx.draw_networkx_nodes(g_kite, pos, nodelist=[5,2], node_color='green')
nx.draw_networkx_nodes(g_kite, pos, nodelist=[4,1], node_color='orange')
nx.draw_networkx_nodes(g_kite, pos, nodelist=[3,0], node_color='purple')
nx.draw_networkx_edges(g_kite, pos=pos)
nx.draw_networkx_labels(g_kite, pos=pos);
```



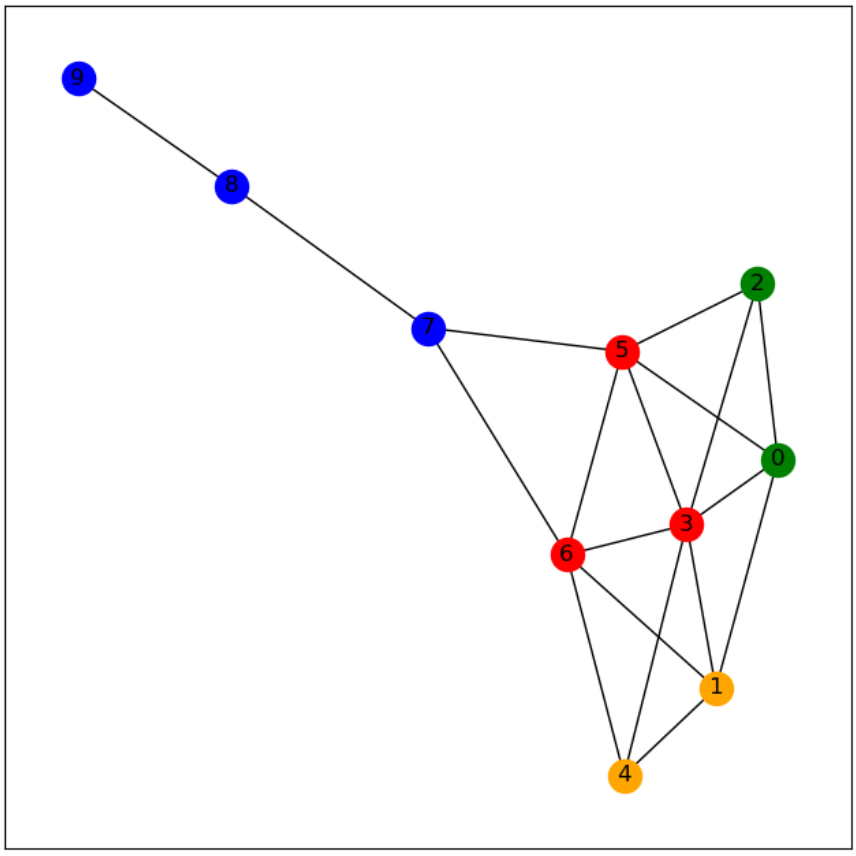

```
nx.algorithms.community.modularity(g_kite, [[9,8], [7,6], [5,2],[4,1],[3,0]])
```

```
0.057098765432098755
```

Then we could keep assigning nodes and see an increase.

```
plt.figure(figsize=(8,8))
pos = nx.spring_layout(g_kite, seed=123)
nx.draw_networkx_nodes(g_kite, pos, nodelist=[9,8,7], node_color='blue')
nx.draw_networkx_nodes(g_kite, pos, nodelist=[6,5,3], node_color='red')
nx.draw_networkx_nodes(g_kite, pos, nodelist=[2,0], node_color='green')
nx.draw_networkx_nodes(g_kite, pos, nodelist=[4,1], node_color='orange')
nx.draw_networkx_edges(g_kite, pos=pos)
```

```
nx.draw_networkx_labels(g_kite, pos=pos);
```

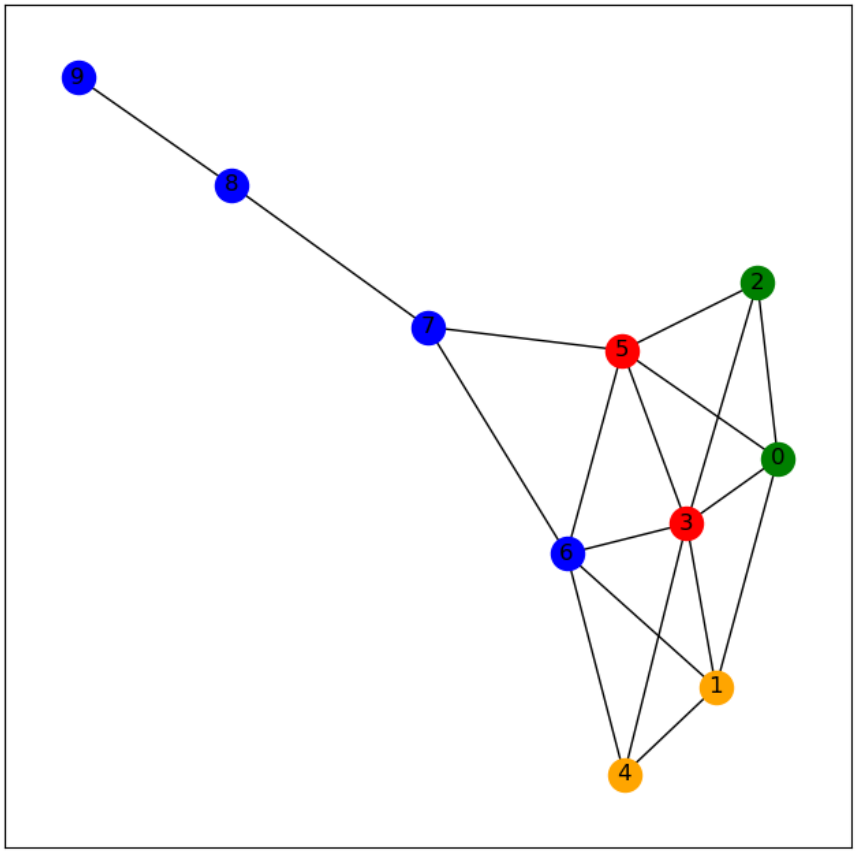



```
nx.algorithms.community.modularity(g_kite, [[9,8,7], [6,5,3], [2,0],[4,1]])
```

```
0.08796296296296295
```

But not every reassignment is going to be an increase. If we extend the first cluster, we will get a decrease in modularity.

```
plt.figure(figsize=(8,8))
pos = nx.spring_layout(g_kite, seed=123)
nx.draw_networkx_nodes(g_kite, pos, nodelist=[9,8,7, 6], node_color='blue')
nx.draw_networkx_nodes(g_kite, pos, nodelist=[5,3], node_color='red')
nx.draw_networkx_nodes(g_kite, pos, nodelist=[2,0], node_color='green')
nx.draw_networkx_nodes(g_kite, pos, nodelist=[4,1], node_color='orange')
nx.draw_networkx_edges(g_kite, pos=pos)
nx.draw_networkx_labels(g_kite, pos=pos);
```




```
nx.algorithms.community.modularity(g_kite, [[9,8,7, 6], [5,3], [2,0],[4,1]])
```

```
0.07098765432098764
```

Our pseudo-Louvain algorithm would therefore stop short of that last assignment. The real Louvain method would do this in an mechanical, incremental fashion until the communities with very good modularity scores are found. The algorithm has elements of randomness, so it is not guaranteed to achieve the best result, or the same result every time it is run.

A difference of note is that instead of incrementally linking up the nodes of the kite, what are the communities at the first level will be turned into nodes at the second level.

What this means in practice, is that unlike what we are doing, the algorithm does not focus on modularity per se, but on modularity gain. That is, it focuses on the difference in modularity between one partitioning and another. It does not really look for the highest modularity value, but instead keeps going while it finds positive modularity gain, or until modularity gains are so small that they fall below a specified margin.

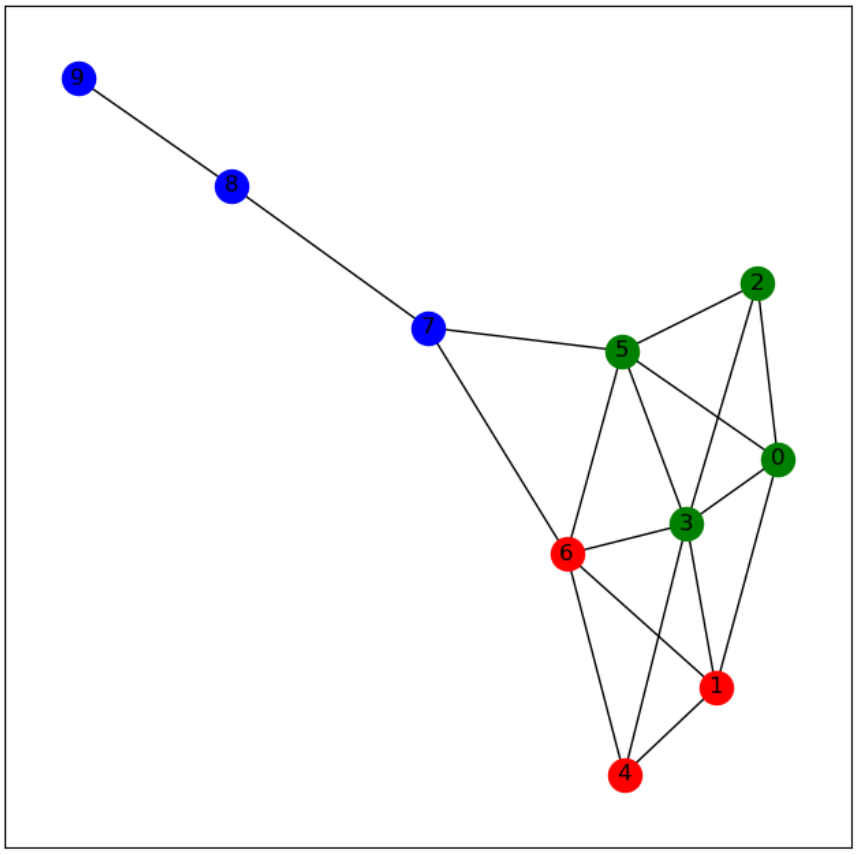
Happily, the formula for modularity gain is not so different from that of modularity

$$\Delta Q = \left[\frac{\Sigma A_c + K_{in}}{2m} - \frac{\Sigma D_c + K_{in}}{2m}\right] - \left[\frac{\Sigma A_c}{2m} - \frac{\Sigma D_c}{2m} - \frac{K_i n}{2m}\right]$$

In fact it is the ratio of two modulary scores. One (the left brackets) is the score with the node K inserted into it and the the other (the right bracket) is the score with the node K removed from it. It is added on the left and removed on the right to represent the delta, or the difference in modularity, when one community gains a node and another one loses it.

With our pseudo-Louvain approach we did not did do the best possible combination, but with the real Louvain communities algorithm we can see what the better partition would be:

```
plt.figure(figsize=(8,8))
pos = nx.spring_layout(g_kite, seed=123)
result = nx.algorithms.community.louvain_communities(g_kite)
nx.draw_networkx_nodes(g_kite, pos, nodelist=result[0], node_color='red')
nx.draw_networkx_nodes(g_kite, pos, nodelist=result[1], node_color='green')
nx.draw_networkx_nodes(g_kite, pos, nodelist=result[2], node_color='blue')
nx.draw_networkx_edges(g_kite, pos=pos)
nx.draw_networkx_labels(g_kite, pos=pos);
```



We can do the same for the subgraph of the CJEU case law example. We first calculate the number of communities:

```
g_consprot2 = load_graph_from_json("data/g_consprot2.json")
communities = sorted(nx.algorithms.community.
 ↪greedy_modularity_communities(g_consprot2), key=len, reverse=True)
print(f"The network has {len(communities)} communities.")
```

```
The network has 13 communities.
```

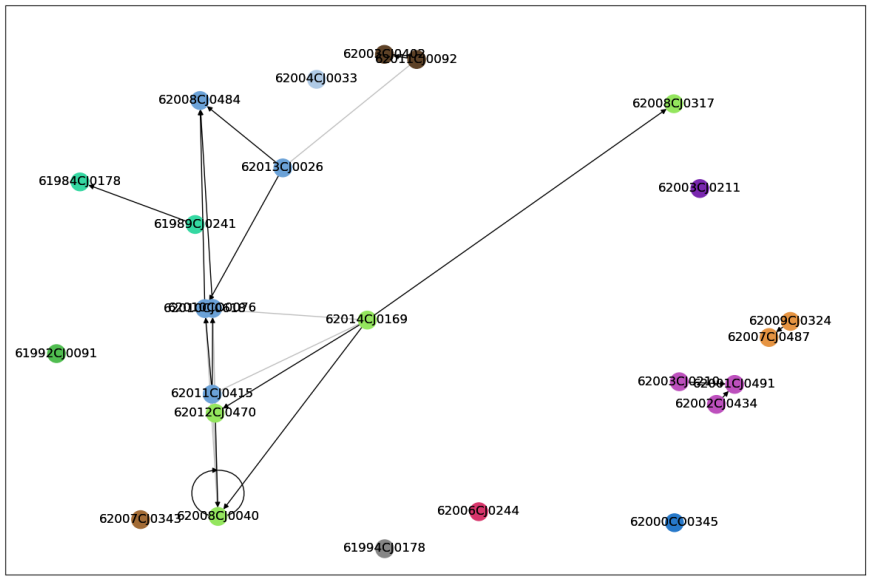
We subsequently plot assign each node to a community and visualize the communities in the network.

```
def set_node_community(G, communities):
    '''Add community to node attributes'''
    for c, v_c in enumerate(communities):
        for v in v_c:
            # Add 1 to save 0 for external edges
            G.nodes[v]['community'] = c + 1
def set_edge_community(G):
    '''Find internal edges and add their community to their attributes'''
    for v, w, in G.edges:
        if G.nodes[v]['community'] == G.nodes[w]['community']:
            # Internal edge, mark with community
            G.edges[v, w]['community'] = G.nodes[v]['community']
        else:
            # External edge, mark as 0
            G.edges[v, w]['community'] = 0
def get_color(i, r_off=1, g_off=1, b_off=1):
    '''Assign a color to a vertex.'''
    r0, g0, b0 = 0, 0, 0
    n = 16
    low, high = 0.1, 0.9
    span = high - low
    r = low + span * (((i + r_off) * 3) % n) / (n - 1)
    g = low + span * (((i + g_off) * 5) % n) / (n - 1)
    b = low + span * (((i + b_off) * 7) % n) / (n - 1)
    return (r, g, b)

# Set node and edge communities
set_node_community(g_consprot2, communities)
set_edge_community(g_consprot2)
node_color = [get_color(g_consprot2.nodes[v]['community']) for v in g_consprot2.
 ↪nodes]
# Set community color for edges between members of the same community (internal)␣
 ↪and intra-community edges (external)
external = [(v, w) for v, w in g_consprot2.edges if g_consprot2.edges[v,␣
 ↪w]['community'] == 0]
internal = [(v, w) for v, w in g_consprot2.edges if g_consprot2.edges[v,␣
 ↪w]['community'] > 0]
internal_color = ['black' for e in internal]
```

```
# Set node and edge communities
set_node_community(g_consprot2, communities)
set_edge_community(g_consprot2)
node_color = [get_color(g_consprot2.nodes[v]['community']) for v in g_consprot2.
 ↪nodes]
# Set community color for edges between members of the same community (internal)␣
 ↪and intra-community edges (external)
external = [(v, w) for v, w in g_consprot2.edges if g_consprot2.edges[v,␣
 ↪w]['community'] == 0]
internal = [(v, w) for v, w in g_consprot2.edges if g_consprot2.edges[v,␣
 ↪w]['community'] > 0]
internal_color = ['black' for e in internal]

g_consprot2_pos = nx.spring_layout(g_consprot2)
plt.rcParams.update({'figure.figsize': (15, 10)})
# Draw external edges
nx.draw_networkx(
    g_consprot2,
    pos=g_consprot2_pos,
    node_size=0,
    edgelist=external,
    edge_color="silver")
# Draw nodes and internal edges
nx.draw_networkx(
    g_consprot2,
    pos=g_consprot2_pos,
    node_color=node_color,
    edgelist=internal,
    edge_color=internal_color)
plt.show()
```



4.2.3 Centrality of communities

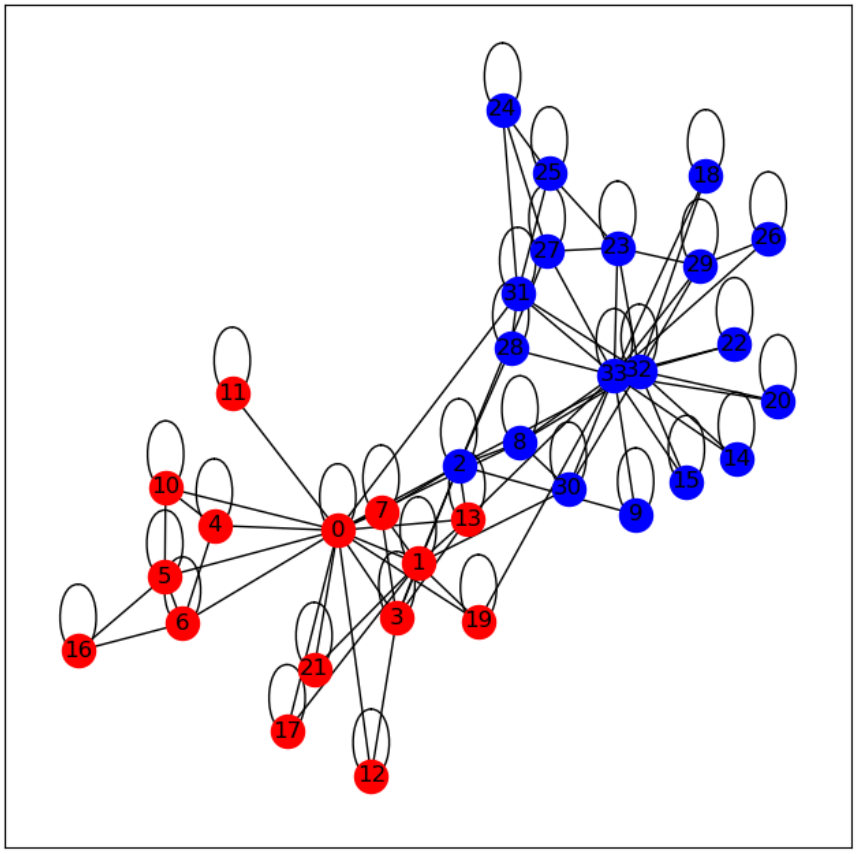
Once a community has been found, it is possible to submit that community to forms of analysis that were first applied to the whole graph. Naturally results will change when we only consider the smaller community.

In NetworkX this is done using the `subgraph` function, passing in the node list of the selected community.

Here we will consider the community that centers around the President of the Karate Club which appears in red below.

```
plt.figure(figsize=(8,8))
pos = nx.spring_layout(g_karate, seed =123)
nx.draw_networkx_nodes(g_karate, pos=pos, nodelist=list(nodes_cluster_1),␣
↪node_color="blue")
nx.draw_networkx_nodes(g_karate, pos=pos, nodelist=list(nodes_cluster_2),␣
↪node_color="red")
nx.draw_networkx_labels(g_karate, pos=pos)
nx.draw_networkx_edges(g_karate, pos=pos)
```

```
<matplotlib.collections.LineCollection at 0x1a64be9c5e0>
```



If we see the closeness centrality scores

```
karate_closeness = pd.DataFrame({"nodes":nx.closeness_centrality(g_karate).
 ↪keys(), "centrality":nx.closeness_centrality(g_karate).values()})\
.sort_values("centrality", ascending=False)\
.head(10)
karate_closeness
```

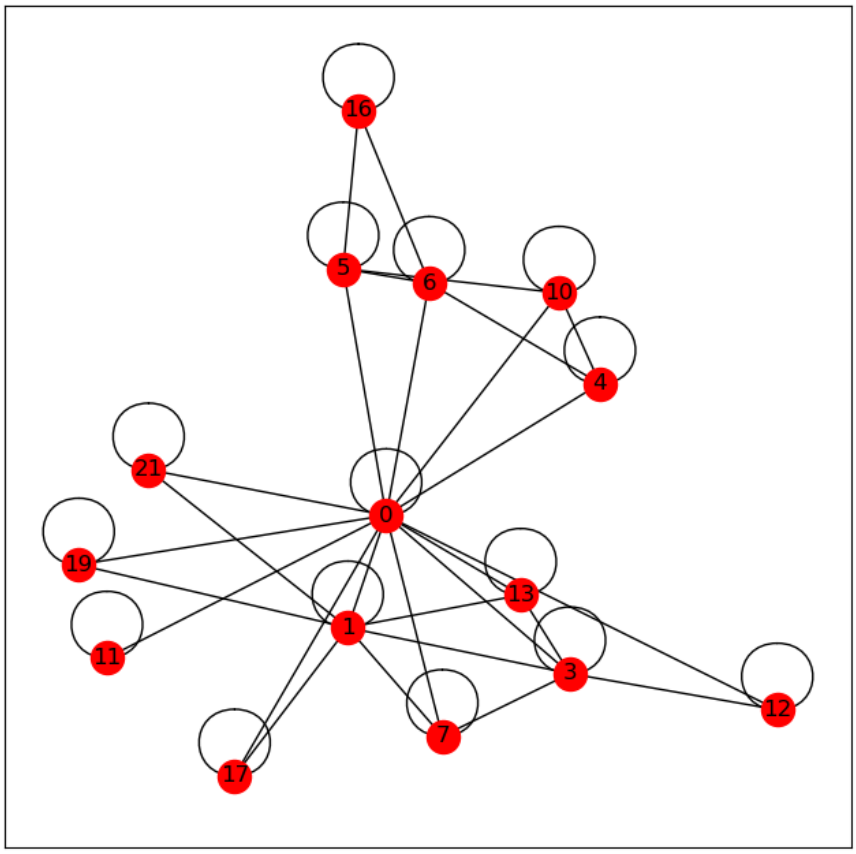
	nodes	centrality
0	0	0.568966
2	2	0.559322
33	33	0.550000
31	31	0.540984
13	13	0.515625
32	32	0.515625
8	8	0.515625
19	19	0.500000
1	1	0.485294
3	3	0.464789

Here we can see that the top performers include the President at node 0, node 2, and the Karate

instructor who is node 33.

However if we consider only the subgraph of the supporters of the President, the network metrics change, including the centrality ranking.

```
g_subkarate = nx.subgraph(g_karate, nodes_cluster_2)
draw_spring(g_subkarate, node_color="red")
```




```
pd.DataFrame({"nodes":nx.closeness_centrality(g_subkarate).keys(), "centrality":
 ↪nx.closeness_centrality(g_subkarate).values()})\
.sort_values("centrality", ascending=False)\
.head(10)
```

	nodes	centrality
0	0	0.933333
1	1	0.636364
2	3	0.583333
4	5	0.583333
5	6	0.583333
3	4	0.560000

```
7       10    0.560000
6        7    0.538462
10      13    0.538462
9       12    0.518519
```

Node 2 is no longer as important now that his friends with the faction of the instructor are factored out. By contrast node 1 rises in prominence.

This analysis is only schematic, but it shows that identifying communities can be an important step for getting more finely grained network metrics.

Chapter 5

Data Collection and Data Preparation

The previous chapters did not yet address a crucial prerequisite for the application of network analysis: ensuring that the data is suitable for conducting network analysis. This chapter addresses this aspect.

The first step is to obtain usable data, and turning the data *into* the network. Obtaining the data may be done by hand, electronically, or both. This data must be put in a format suitable to be read and processed by network analysis software. In this respect, we will discuss how data might be extracted through qualitative coding or data mining. We will focus on the ways you can structure the data in a format that is readable by the software. What you will see here are just examples, as the researcher will need to develop a strategy that fits his / her own problem.

5.1 The Research Question Offers Guidance

What data is required, first and foremost depends on the aim of the research and the questions that you are asking. These determine the design and the scope of the research and therefore affect the boundaries of the network that you are investigating. The research questions will focus on certain relations (edges), offering guidance as to what elements (nodes) or variables to include in the network and which in turn determines the data that you need to obtain. For instance, if we want to conduct a network analysis of an organized crime syndicate, a number of questions could be relevant. We can ask which actors are related to each other and who is the most important actor given those relations? Moreover, we could examine the types of criminal offences they were involved in, when, and with whom? This helps us to answer the broader question how criminal networks and activities develop over time. We could also include metadata about gender, age or nationality of the actor and locus delicti of the crimes, etc. to shed even more light on the characteristics of the offender and the offences.

To develop such a network analysis of an organized crime syndicate you can rely on information that can be retrieved from various sources, such as newspaper articles that report on the activities of the network, or more elaborate journalistic and scholarly investigations that address the criminal network and its characteristics more in depth. Or maybe you can even get access to police or court files that shed light on the criminal activities, their organization and the question to what extent these activities amount to criminal offences.

Our first question concerns a straightforward actor network, which means that we will need to analyse and code our journalistic, scholarly, police or court data for actors and their relations. To address our second question, we will need to go through our data and we need to code the offences perpetrated, the date and location, and the names of the actors involved, etc.

5.2 Data Collection and Qualitative Coding

To obtain the relevant data you will need to look for data sources, which can include (online) communities, events, actors, archives, etc. To extract relevant data from these sources, researchers often rely on data collection methods used in the social sciences such as field observations, open or structured interviews, surveys, archival and documentation research, etc. The collected data might already be suitable for network analysis, because the data is for instance numerical, which can be the case when doing surveys. However, the data gathered will often probably be qualitative in nature. It will concern written documentation, archival records, interview transcripts, photographs, video footage, field study reports. memo's etc. In order for such qualitative data to be used, it needs to be formatted into data that can be used in network analysis. This can be done manually through qualitative coding or annotation. This involves the process of defining / labelling the data according to key themes. These categories can be distilled inductively 'in vivo' from the data itself or the categories can be taken from relevant theoretical and conceptual frameworks.

One may examine corporate reports and interviews, and find evidence of partnership relations between corporations. Some of this evidence may be formal statements of partnership. Other evidence may be testiominal evidence to the effect that "we always buy equipment from Nissan". Regarding the second example, we may rely on attitudes expressed in formal written exchanges or in interviews with diplomats. At any rate this information will be qualitative, and thus the researcher will need to judge whether evidence such as "we always buy equipment from Nissan" is sufficient grounds to say that there is a collaboration, and "code" it as such, or not. Or, alternatively, the researcher will want to "weigh" the strength of the evidence, the strength of collaboration, or both, leading to networks with weighted edges. This process will almost always involve value judgments on the part of the researcher. One of the ways that these value judgments can be rendered more objective is through inter-coder reliability. That means that more than one researcher will code (for example, a team of students) and then findings will be weighted down by the degree of disagreement there is between coders.

Qualitative coding thus refers to the process of extracting categorized information from qualitative data sources. Note that this "coding" does not have anything to do with "programming". Coding here means "encoding" the qualitative information into headings or categories that can later be counted and related to each other, for example through networks. Software such as NVivo or Atlas.ti can help you to archive and manage your data and also assist in manual qualitative coding.

Instead of or in addition to manual coding, qualitative data can, if this is appropriate, be automatically processed and parsed into a meaningful format using a variety of tools. For example, for qualitative text data, methods such as Regular Expressions or Natural Language Processing (NLP) can be used to "mine" qualitative sources, without doing qualitative coding. A relevant example is a network of document similarity. Imagine one starts with a corpus of legally relevant documents, for example a corpus of stakeholder contributions to a regulation. NLP software can produce a similarity score for every pair of such documents. The documents can then be put into a network,

with links for documents with high similarity. This can be used to see which were the central or paradigmatic contributions to the negotiation process.

It should, however, not be concluded that this type of data mining is free of value judgments. For example, one may need to decide on the similarity threshold. At what degree of similarity is a line drawn between two documents? 0.7? 0.8? 0.9? Different thresholds will lead to different results (naturally one can also do away with the thresholds and just get the similarity as a weigth, but that would also be a research decision needing justification).

5.3 Data Representation

Let us assume that one way or another, we have settled on a strategy to obtain the data we need. But this data needs to be recorded in a way that Network Software can read it. There are three main options for doing this. We will demonstrate how to represent graph data as an adjacency matrix, in tabular/CSV format, and in JSON format.

Since networks capture relations, what qualitative coding practices must do is classify relations between subjects of study. Looking back at Chapter 1, we can see that this can be between subjects of the same type, or between subjects of different types, in the case of bipartite networks. For an example of the first scenario, consider research trying to identify which big corporations work together in a particular country. An example of the second scenario concerns research trying to establish which states have strong ties to a particular international organization, such as the WTO or the UN.

Such networks are collections of nodes that are connected by edges. The most basic way to conceptualize such a network is using a matrix encompassing the relational ties between the nodes. As shown in previous chapters, such an adjacency matrix is an array or list of data that numerically organizes / indexes the data in rows and columns to capture the relationships. The simplest and most common matrix is the binary matrix that numerically indicates whether a relationship or connection exists or not between the nodes. If a tie is present, a one is entered in a cell; if there is no tie, a zero is entered. An adjacency matrix is a $n \times n$ matrix that relates the entities in the rows, to the entities in the columns. Because every entity is related to every other entity, the matrix will always be a square one, it will always have the same number of rows and columns. This is in fact implied in the notation $n \times n$; the same letter is used, so if n = 5, the matrix is 5 x 5, if n = 10 it is 10 x 10, etc., as opposed to an $m \times n$ matrix where it can be the case that m = 5, and n = 4, or something similar.

Every "1" in the adjacency matrix will show the presence of a particular relationship, while every "0" will note its absence. A 4×4 adjacency matrix might look like this.

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

We can use the adjacency matrix to enter and represent the data regarding the actors involved in the criminal network discussed before. Naturally the easiest way to get data into this structure is through a spreadsheet program. For instance, from our information it becomes clear that 14

individuals show up in the data that we can relate to each other. The relationships can be put in a data frame or matric as shown below. A DataFrame is a generic name for a data strucutre that has columns for variables and rows for observations, allowing, optionally, a first row for variable names. Any spreadsheet program can be used to make a DataFrame.

	Sam de Klerk	Karel de Wit	Bruna van Dam	Rachel Gielissen	Dirk van Hattum	John Newman	Roland Vasquez	Peter Vogelaar	Niels Boom	Randy van Wijk	Sjaak Zwart	Klaas de Bruijn	Charles Doomen	Nico Doomen
Sam de Klerk	1	1	0	1	0	1	0	1	1	1	1	1	0	0
Karel de Wit	1	0	1	1	0	1	0	1	1	0	0	0	1	1
Bruna van Dam	0	1	0	0	1	1	0	1	1	1	1	1	0	0
Rachel Gielissen	1	1	0	0	0	1	0	0	0	1	1	1	1	1
Dirk van Hattum	1	1	1	0	0	1	0	0	0	0	1	1	1	1
John Newman	1	1	1	1	1	0	1	1	1	1	1	1	1	1
Roland Vasquez	1	1	1	0	1	1	0	1	1	1	1	1	1	1
Peter Vogelaar	1	0	1	0	1	1	0	0	1	0	0	0	1	1
Niels Boom	1	1	0	0	1	1	0	1	0	1	0	1	0	1
Randy van Wijk	1	1	0	0	0	1	0	1	1	0	1	1	0	0
Sjaak Zwart	1	1	1	0	1	1	0	0	0	1	0	1	1	1
Klaas de Bruijn	1	0	0	0	1	1	0	1	1	1	0	0	1	1
Charles Doomen	1	1	1	0	0	1	0	0	0	1	1	1	0	1
Nico Doomen	1	1	1	1	1	1	0	1	1	1	1	1	1	0

If the network links have specific weights, these will replace the 1s and 0s in the Spreadsheet.

$$M = \begin{pmatrix} 0 & 0.5 & 0.5 & 0 \\ 0.5 & 0 & 1 & 1 \\ 0.5 & 1 & 0 & 0.2 \\ 0 & 1 & 0.2 & 0 \end{pmatrix}$$

Additional attribute data can be included in the matrix data structure by adding keeping a separate list of a nodes, with the attribute associated with it. For example, the column of names of the spreadsheet above, with a column denoting their gender adjacent to it.

A, Male

B, Female

C, Female

D, Male

An alternative representation can be a so-called edge list, which includes a set of rows in which each row represents a specific tie in the overall network. An edge list consists of sources and targets, which, in a directed graph, coincide with the source nodes and target nodes. In a network with four nodes (A,B,C,D) where Node A is connected to Node B and Node D, where Node B is connected to Node A and Node C, and Node C is connected to Node B, an edge list will take the following shape:

A, B

A, D

B, A

B, C

C, B

If the network is undirected the above list contains redundant information. The link "C, B" adds nothing to the already present link "B, C") and thus can be deleted.

For weighted edges, the weight can follow the specification of the node pairs.

A, B, 0.5

A, D, 0.2

B, A, 0.5

B, C, 1

C, B, 1

Here once more, attribute data can be collected in a separate list of nodes, with columns for attributes.

Chapter 6

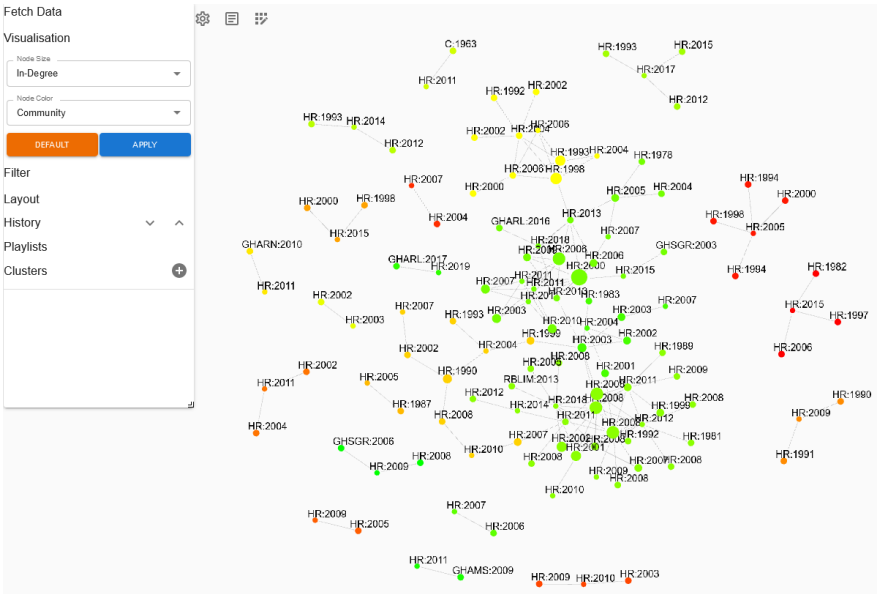
Software

By now, you are almost ready to start conducting legal network analysis yourself. But what software to use? There are essentially three options: 1. Network analysis software designed for a specific purpose. 2. General-purpose software. 3. Packages in programming software (e.g. Python, R).

6.1 Specific Purpose Network Analysis Software

Software has been developed for specific use cases. The advantage of such software is that it is optimized for that use case and that it requires a low level of technical knowledge in order to use the software. The disadvantage is that the platform is designed for a specific purpose, which restricts the freedom in terms of adding or removing functionality.

An example of this type of software concerns Case Law Explorer. This platform has been developed for students, researchers, and practitioners who wish to conduct network analysis on case law. It has automated the process from searching to the calculation of the network statistics and the visualization of the network. A link to the software and to the user guide of the software can be found here.



6.2 General Purpose Network Analysis Software

General purpose software allows applying network analysis to different types of data. The advantage of this type of software is that it can be applied in a number of contexts and for a variety of purposes, whereas the disadvantage is that the functionalities are generic and not tailored to a specific use case.

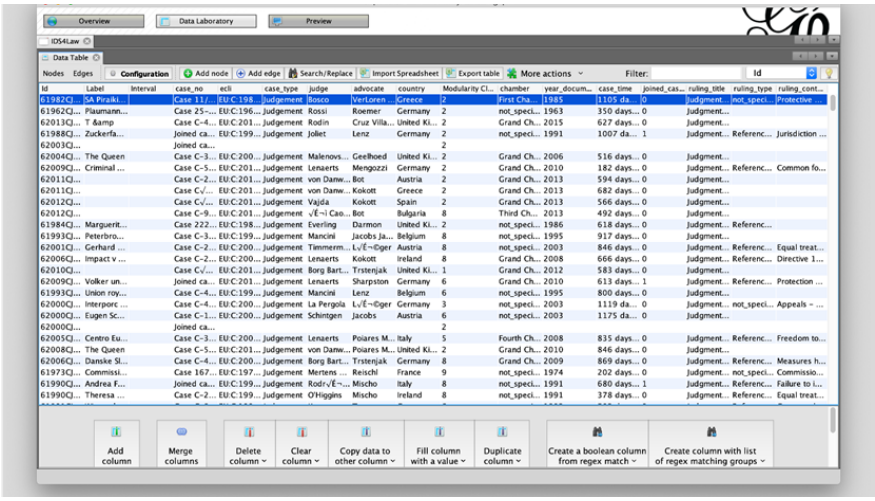
6.2.1 Gephi

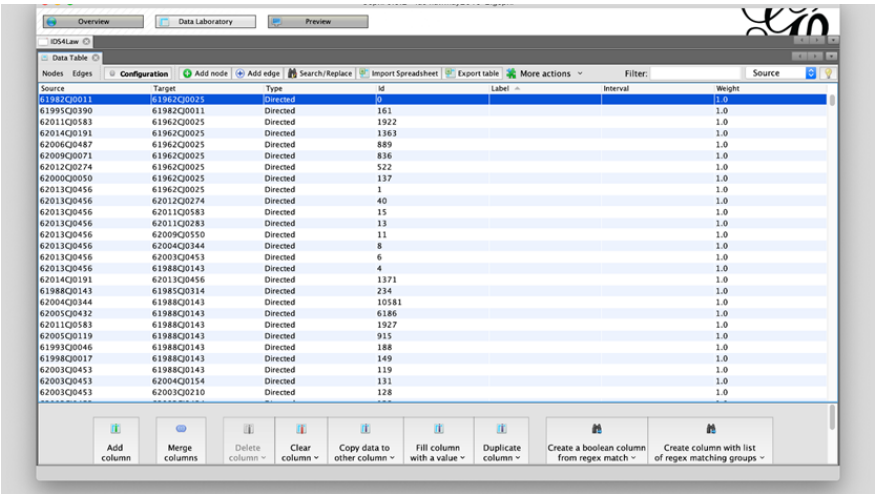
Gephi is perhaps the easiest to use menu driven software package for network analysis.

Preliminaries. You need to download and install Gephi on your laptop. In case of a Java error when opening Gephi, update Java. Check this video (Windows) if Gephi still returns a Java error.

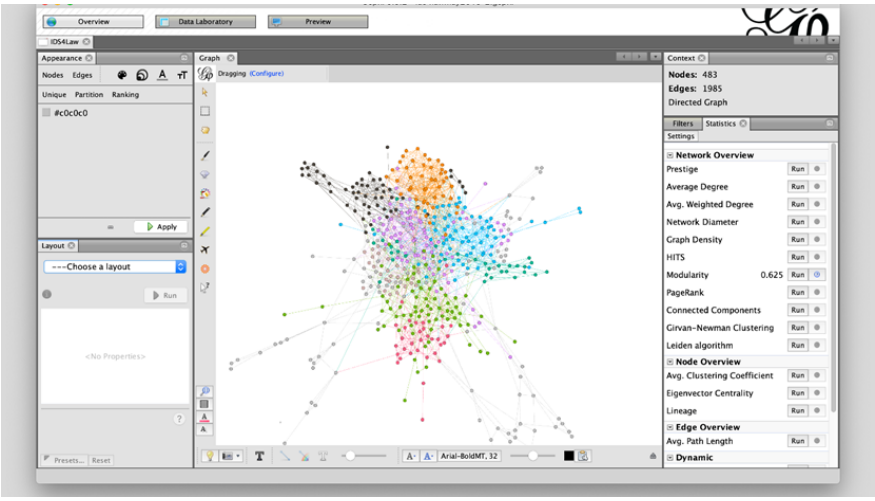
Navigating Gephi. Gephi has three main modes: Data Laboratory, Overview and Preview.

- Data Laboratory. This is where you can import data and view network data in table format, split into Nodes and Edges. There is a limit on the number of columns this tab will display. Click the light bulb icon on the top right part of the window to choose which columns you want to see.

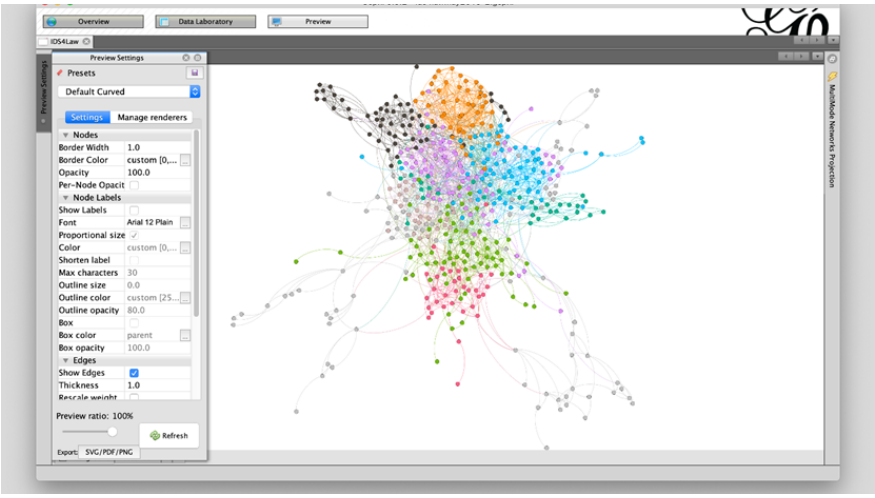




- Overview. This is where the magic happens! The Overview tab has four main windows.
  - *Appearance* (upper left) is used to control node and edge presentation properties.
  - *Layout* (lower left) is behind the network layout.
  - *Graph* (centre) shows the sketch of the graph.
  - *Statistics & Filters* (right) is for network analysis and filtering. In the *Statistics* panel, you can calculate various network statistics. The *Filters* panel supports complex methods to temporarily highlight or hide a part of the network.



- Preview. This is where you prepare your final image for export. To see it displayed in the screen click on *Refresh* at the bottom of the panel.



Importing the Data. Given that you have an edges list (in csv) and possibly a nodes list (in csv), you can follow the following steps in order to import the data: 1. Start Gephi
2. From the dialog that appears after startup, or from the file menu, start a New Project 3. Go to Data Laboratory 4. Choose Import Spreadsheet 5. Select your csv file with the nodes 6. Set the right separator (comma) and set as 'nodes table' 7. Click Next, then Finish 8. Choose to 'append to existing workspace' and click Ok. You should see a neat nodes table now 9. Click Import Spreadsheet again 10. Repeat the process for the file with edges, but this time choose 'edges table' as the table type 11. Click Next, then Finish 12. Choose to 'append to existing workspace' and click Ok

Investigating the network. The next step is to investigate the network. To get started: 1. Go to the Overview tab. You should now see your network. 2. Run the ForceAtlas 2 layout algorithm in Layout. Note that every layout algorithm can be tweaked by changing parameters. You can play around with the 'Scaling' and 'Prevent overlap' parameters.

We subsequently calculate the centrality statistics, 3. Calculate the centralities by running 'Average

degree' in the Statistics tab. Gephi will show you the degree distributions, but will also add the results to the dataset in Data Laboratory. You may also run 'Network Diameter', which gives you various centrality measures based on network diameter, including betweenness centrality.

change the node size based on the centrality measure of our preference, 4. We visualize the nodes with a higher in-degree by increasing their size. Click on 'Nodes' in the Appearance tab, then click the  (size) button and choose 'Ranking'. Select 'In-Degree' from the drop-down attribute menu and click Apply.

and filter the network. 5. You can, for instance, visualize the network of the 50 most central cases (based on in-degree). Go to Data Laboratory and determine the in-degree of the 50th most central case. Then go back to Overview and open the Filters window. Expand the Attributes filter category and select 'Range', then double click on 'In-Degree'. Finally, adjust the indegree range and click Filter. Click Stop to proceed with the next part.

We can also calculate the community structure, 6. Run 'Modularity' in the Statistics tab.

adjust the appearance, 7. Recolor the communities by clicking on 'Nodes' in the Appearance tab, then click the  (color) button and choose 'Partition'. Select 'Modularity Class' from the drop-down attribute menu and click Apply.

and filter for certain communities. 8. Choose a community and layout again. Open the Filters window. Expand the Partition filter category and double click on 'Modularity Class'. Finally, select a community to display and click Filter.

This is only a very basic introduction to Gephi. More information and many videos can be found online (eg here), and we gladly refer to those instructions and materials.

(Note: credits to Constanta Rosca, who co-drafted these instructions for a previously held workshop.)

6.2.2 Other Software Packages

Other software packages that may be used for conducting network analysis include:

- Pajek
- UCINET
- Cytoscape
- SocNetV

6.3 Packages in programming software (e.g. Python, R)

Another option is to use packages such as *NetworkX* or *iGraph* that can be installed and imported in programming languages such as *Python* and *R*. These packages provide a user substantial flexibility as to what the manipulation and calculation of the data. Using the packages does require the knowledge of programming language (or the willingness to learn how to code), and it can be labor-intensive to have to write out all the steps in lines of code.